MASDG

Mobile Application Security Design Guide for Android

Yoshiaki Yasuda

OWASP®

Contents

April 1, 2023 Edition

Welcome to Mobile Application Security Design Guide Android Edition.

The Mobile Application Security Design Guide Android Edition is a document aimed at establishing a framework for designing, developing, and testing secure mobile applications on Android, incorporating our own evaluation criteria (rulebook) and sample code into the OWASP Mobile Application Security Verification Standard (MASVS) and Mobile Application Security Testing Guide (MASTG) published by OWASP.

MASDG deals with best practices and samples that are specific to the design requirements for security, supporting the creation of security designs from security requirements considered based on MASVS, as well as evaluating the security design for any issues before conducting testing methods indicated in MASTG.

Our proprietary rulebook targets the MASVS L1 verification standard and aims to provide comprehensive security baselines when developing mobile applications. While new technologies will always bring risks and create privacy and safety issues, we have created this document to address the threats posed by mobile applications.

We have received feedback on MASVS and MASTG from various communities and industries, and we have developed and published MASDG as we believe it is essential to tackle the security risks associated with mobile applications that have become indispensable in our society. We welcome feedback from everyone.

Copyright and License

- The contents of this guide are current as of the time of writing. Please be aware of this if you use the sample code.
- The authors are not responsible for any consequences resulting from the use of this guide. Please use at your own risk.
- Android is a trademark or registered trademark of Google LLC. Company names, product names, and service names mentioned in this document are generally registered trademarks or trademarks of their respective companies. The ®, TM, and © symbols are not used throughout this document.
- Some of the content in this document is based on the materials provided by OWASP MASVS and OWASP MASTG, and has been replicated and revised.

Originator
Project Site - https://owasp.org/www-project-mobile-app-security/
Project Repository - https://github.com/OWASP/www-project-mobile-app-security
MAS Official Site - https://mas.owasp.org/
MAS Document Site - https://mas.owasp.org/MASVS/
MAS Document Site - https://mas.owasp.org/MASTG/
Document Site - https://mobile-security.gitbook.io/masvs
Document Repository - https://github.com/OWASP/owasp-masvs
Document Site - https://coky-t.gitbook.io/owasp-masvs-ja/
Document Repository - https://github.com/owasp-ja/owasp-masvs-ja
Document Site - https://mobile-security.gitbook.io/mobile-security-testing-guide
Document Repository - https://github.com/OWASP/owasp-mastg
Document Site - https://coky-t.gitbook.io/owasp-mastg-ja/
Document Repository - https://github.com/coky-t/owasp-mastg-ja

OWASP MASVS Authors

Project Lead	Lead Author	Contributors and Reviewers
Sven Schleier, Carlos Holguera	Bernhard Mueller, Sven Schleier, Jeroen Willemsen and Carlos Holguera	Alexander Antukh, Mesheryakov Aleksey, Elderov Ali, Bachevsky Artem, Jeroen Beckers, Jon-Anthoney de Boer, Damien Clochard, Ben Cheney, Will Chilcutt, Stephen Corbiaux, Manuel Delgado, Ratchenko Denis, Ryan Dewhurst, @empty_jack, Ben Gardiner, Anton Glezman, Josh Grossman, Sjoerd Langkemper, Vinícius Henrique Marangoni, Martin Marsicano, Roberto Martelloni, @PierrickV, Julia Potapenko, Andrew Orobator, Mehrad Rafii, Javier Ruiz, Abhinav Sejpal, Stefaan Seys, Yogesh Sharma, Prabhant Singh, Nikhil Soni, Anant Shrivastava, Francesco Stillavato, Abdessamad Temmar, Pauchard Thomas, Lukasz Wierzbicki

OWASP MASTG Authors
Bernhard Mueller
Sven Schleier
Jeroen Willemsen
Carlos Holguera
Romuald Szkudlarek
Jeroen Beckers
Vikas Gupta

OWASP MASVS ja Author
Koki Takeyama

OWASP MASTG ja Author
Koki Takeyama

Project Supporter
Riotaro Okada

1 Architecture, Design and Threat Modeling Requirements

1.1 MSTG-ARCH-1

All app components are identified and known to be needed.

1.1.1 Component

Get to know all the components of your application and remove unnecessary ones.

The main types of components are as follows.

- Activity
- Fragment
- Intent
- BroadcastReceiver
- ContentProvider
- Service

1.2 MSTG-ARCH-2

Security controls are never enforced only on the client side, but on the respective remote endpoints.

1.2.1 Falsification of Authentication/Authorization information

1.2.1.1 Appropriate authentication response

Perform the following steps when testing authentication and authorization.

- Identify the additional authentication factors the app uses.
- Locate all endpoints that provide critical functionality.
- Verify that the additional factors are strictly enforced on all server-side endpoints.

Authentication bypass vulnerabilities exist when authentication state is not consistently enforced on the server and when the client can tamper with the state. While the backend service is processing requests from the mobile client, it must consistently enforce authorization checks: verifying that the user is logged in and authorized every time a resource is requested.

Consider the following example from the OWASP Web Testing Guide. In the example, a web resource is accessed through a URL, and the authentication state is passed through a GET parameter:

```
http://www.site.com/page.asp?authenticated=no
```

The client can arbitrarily change the GET parameters sent with the request. Nothing prevents the client from simply changing the value of the authenticated parameter to "yes", effectively bypassing authentication.

Although this is a simplistic example that you probably won't find in the wild, programmers sometimes rely on "hidden" client-side parameters, such as cookies, to maintain authentication state. They assume that these parameters can't be tampered with. Consider, for example, the following classic vulnerability in Nortel Contact Center Manager. The administrative web application of Nortel's appliance relied on the cookie "isAdmin" to determine whether the logged-in user should be granted administrative privileges. Consequently, it was possible to get admin access by simply setting the cookie value as follows:

```
isAdmin=True
```

Security experts used to recommend using session-based authentication and maintaining session data on the server only. This prevents any form of client-side tampering with the session state. However, the whole point of using stateless authentication instead of session-based authentication is to not have session state on the server. Instead, state is stored in client-side tokens and transmitted with every request. In this case, seeing client-side parameters such as isAdmin is perfectly normal.

To prevent tampering cryptographic signatures are added to client-side tokens. Of course, things may go wrong, and popular implementations of stateless authentication have been vulnerable to attacks. For example, the signature verification of some JSON Web Token (JWT) implementations could be deactivated by setting the signature type to "None". We'll discuss this attack in more detail in the "Testing JSON Web Tokens" chapter.

Reference

- owasp-mastg Verifying that Appropriate Authentication is in Place (MSTG-ARCH-2 and MSTG-AUCH-1)

1.2.2 Injection Flaws

An injection flaw describes a class of security vulnerability occurring when user input is inserted into backend queries or commands. By injecting meta-characters, an attacker can execute malicious code that is inadvertently interpreted as part of the command or query. For example, by manipulating a SQL query, an attacker could retrieve arbitrary database records or manipulate the content of the backend database.

Vulnerabilities of this class are most prevalent in server-side web services. Exploitable instances also exist within mobile apps, but occurrences are less common, plus the attack surface is smaller.

For example, while an app might query a local SQLite database, such databases usually do not store sensitive data (assuming the developer followed basic security practices). This makes SQL injection a non-viable attack vector. Nevertheless, exploitable injection vulnerabilities sometimes occur, meaning proper input validation is a necessary best practice for programmers.

Reference

- owasp-mastg Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2)

Rulebook

- *Enforce appropriate input validation (Required)*

1.2.2.1 SQL Injection

A SQL injection attack involves integrating SQL commands into input data, mimicking the syntax of a predefined SQL command. A successful SQL injection attack allows the attacker to read or write to the database and possibly execute administrative commands, depending on the permissions granted by the server.

Apps on both Android and iOS use SQLite databases as a means to control and organize local data storage. Assume an Android app handles local user authentication by storing the user credentials in a local database (a poor programming practice we' ll overlook for the sake of this example). Upon login, the app queries the database to search for a record with the username and password entered by the user:

```
SQLiteDatabase db;

String sql = "SELECT * FROM users WHERE username = '" +  username + "' AND␣
↪password = '" + password +"'";

Cursor c = db.rawQuery( sql, null );

return c.getCount() != 0;
```

Let' s further assume an attacker enters the following values into the "username" and "password" fields:

```
username = 1' or '1' = '1
password = 1' or '1' = '1
```

This results in the following query:

```
SELECT * FROM users WHERE username='1' OR '1' = '1' AND Password='1' OR '1' = '1'
```

Because the condition '1' = '1' always evaluates as true, this query return all records in the database, causing the login function to return true even though no valid user account was entered.

Ostorlab exploited the sort parameter of Yahoo' s weather mobile application with adb using this SQL injection payload.

Another real-world instance of client-side SQL injection was discovered by Mark Woods within the "Qnotes" and "Qget" Android apps running on QNAP NAS storage appliances. These apps exported content providers vulnerable to SQL injection, allowing an attacker to retrieve the credentials for the NAS device. A detailed description of this issue can be found on the Nettitude Blog.

Reference

- owasp-mastg Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2) SQL Injection

1.2.2.2 XML Injection

In a XML injection attack, the attacker injects XML meta-characters to structurally alter XML content. This can be used to either compromise the logic of an XML-based application or service, as well as possibly allow an attacker to exploit the operation of the XML parser processing the content.

A popular variant of this attack is XML eXternal Entity (XXE). Here, an attacker injects an external entity definition containing an URI into the input XML. During parsing, the XML parser expands the attacker-defined entity by accessing the resource specified by the URI.

The integrity of the parsing application ultimately determines capabilities afforded to the attacker, where the malicious user could do any (or all) of the following: access local files, trigger HTTP requests to arbitrary hosts and ports, launch a cross-site request forgery (CSRF) attack, and cause a denial-of-service condition. The OWASP web testing guide contains the following example for XXE:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

In this example, the local file /dev/random is opened where an endless stream of bytes is returned, potentially causing a denial-of-service.

The current trend in app development focuses mostly on REST/JSON-based services as XML is becoming less common. However, in the rare cases where user-supplied or otherwise untrusted content is used to construct XML queries, it could be interpreted by local XML parsers, such as NSXMLParser on iOS. As such, said input should always be validated and meta-characters should be escaped.

Reference

- owasp-mastg Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2) XML Injection

Rulebook

- *Input should always be validated and meta-characters should be escaped (Required)*

1.2.2.3 Injection Attack Vectors

The attack surface of mobile apps is quite different from typical web and network applications. Mobile apps don't often expose services on the network, and viable attack vectors on an app's user interface are rare. Injection attacks against an app are most likely to occur through inter-process communication (IPC) interfaces, where a malicious app attacks another app running on the device.

Locating a potential vulnerability begins by either:

- Identifying possible entry points for untrusted input then tracing from those locations to see if the destination contains potentially vulnerable functions.
- Identifying known, dangerous library / API calls (e.g. SQL queries) and then checking whether unchecked input successfully interfaces with respective queries.

During a manual security review, you should employ a combination of both techniques. In general, untrusted inputs enter mobile apps through the following channels:

- IPC calls
- Custom URL schemes
- QR codes
- Input files received via Bluetooth, NFC, or other means
- Pasteboards
- User interface

Verify that the following best practices have been followed:

- Untrusted inputs are type-checked and/or validated using a list of acceptable values.
- Prepared statements with variable binding (i.e. parameterized queries) are used when performing database queries. If prepared statements are defined, user-supplied data and SQL code are automatically separated.
- When parsing XML data, ensure the parser application is configured to reject resolution of external entities in order to prevent XXE attack.
- When working with X.509 formatted certificate data, ensure that secure parsers are used. For instance Bouncy Castle below version 1.6 allows for Remote Code Execution by means of unsafe reflection.

We will cover details related to input sources and potentially vulnerable APIs for each mobile OS in the OS-specific testing guides.

Reference

- owasp-mastg Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2) Injection Attack Vectors

Rulebook

- *Do not include potentially vulnerable functions in the destination (Required)*
- *Unchecked inputs are successfully linked to their respective queries (Required)*

- *Check for untrusted input (Required)*
- *Parser application is configured to refuse to resolve external entities (Required)*
- *Use a secure parser when using certificate data in X.509 format (Required)*

1.2.3 Rulebook

1. *Enforce appropriate input validation (Required)*
2. *Input should always be validated and meta-characters should be escaped (Required)*
3. *Do not include potentially vulnerable functions in the destination (Required)*
4. *Unchecked inputs are successfully linked to their respective queries (Required)*
5. *Check for untrusted input (Required)*
6. *Parser application is configured to refuse to resolve external entities (Required)*
7. *Use a secure parser when using certificate data in X.509 format (Required)*

1.2.3.1 Enforce appropriate input validation (Required)

Proper input validation is a necessary best practice for programmers, as injection vulnerabilities can be exploited.

Below is an example of input validation.

- Regular expression check
- Length/size check

If this is violated, the following may occur.

- An injection vulnerability may be exploited.

1.2.3.2 Input should always be validated and meta-characters should be escaped (Required)

If an XML query is created using user-supplied or untrusted content, XML metacharacters may be interpreted as XML content by the local XML parser.Therefore, input should always be validated and meta-characters should be escaped.

Below is an example of input validation.

- Regular expression check
- Length/size check

Below is an example of a meta-character.

Table 1.2.3.2.1 List of Meta-Characters

Character	Item Name	Entity Reference Notation
<	Right Greater Than	&lt;
>	Left Greater Than	&gt;
&	Ampersand	&amp;
“	Double Quotation	&quot;
‘	Single Quotation	&apos;

If this is violated, the following may occur.

- XML meta characters may be interpreted as XML content by the local XML parser.

1.2.3.3 Do not include potentially vulnerable functions in the destination (Required)

Identify potential entry points for untrusted inputs and trace from that location to see if the destination contains potentially vulnerable functions (third-party functions).

If this is violated, the following may occur.

- A malicious app could attack another app running on the device via the Inter-Process Communication (IPC) interface.

1.2.3.4 Unchecked inputs are successfully linked to their respective queries (Required)

Identify known dangerous library /API calls (e.g., SQL queries) and verify that unchecked inputs work with the respective queries successfully. Also, check the reference of the library/API to be used and confirm that it is not deprecated.

Android API Reference：https://developer.android.com/

If this is violated, the following may occur.

- A malicious app could attack another app running on the device via the Inter-Process Communication (IPC) interface.

1.2.3.5 Check for untrusted input (Required)

In general, untrusted inputs enter mobile apps through the following channels:

Below is an example of keywords that identify channel use.

- IPC calls：ContentProvider
- Custom URL schemes：scheme
- QR codes：qr, camera
- Input files received via Bluetooth, NFC, or other means：bluetoothAdapter
- Pasteboards：ClipboardManager
- User interface：EditText

If this is violated, the following may occur.

- A malicious app could attack another app running on the device via the Inter-Process Communication (IPC) interface.

1.2.3.6 Parser application is configured to refuse to resolve external entities (Required)

To prevent XXE attacks, parser applications should be configured to refuse to resolve external entities.

The safest way to prevent XXE is always to disable DTDs (External Entities) completely. Depending on the parser, the method should be similar to the following:

```
factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
```

Disabling DTD also makes the parser secure against denial of services (DOS) attacks such as Billion Laughs. If it is not possible to disable DTDs completely, then external entities and external document type declarations must be disabled in the way that' s specific to each parser.

If this is violated, the following may occur.

- Be vulnerable to XXE attacks.

Reference

- OWASP Cheat Sheet Series XML External Entity Prevention

1.2.3.7 Use a secure parser when using certificate data in X.509 format (Required)

Use a secure parser when handling X.509 format certificate data.

Below is an example of a safe parser.

- Bouncy Castle (Version 1.6 or higher)

If this is violated, the following may occur.

- Vulnerable to injection attacks, such as remote code execution via insecure reflection.

1.3 MSTG-ARCH-3

A high-level architecture for the mobile app and all connected remote services has been defined and security has been addressed in that architecture.

* Guide description is omitted in this document as this chapter is about support on the remote service side.

1.4 MSTG-ARCH-4

Data considered sensitive in the context of the mobile app is clearly identified.

Classifications of sensitive information differ by industry and country. In addition, organizations may take a restrictive view of sensitive data, and they may have a data classification policy that clearly defines sensitive information. If a data classification policies is not available, use the following list of information generally considered sensitive.

Reference

- owasp-mastg Penetration Testing (a.k.a. Pentesting) Identifying Sensitive Data

Rulebook

- *Identify sensitive data according to data classification policies (Required)*

1.4.1 User authentication information

User authentication information (credentials, PIN, etc.).

1.4.2 Personally Identifiable Information

Personally Identifiable Information (PII) that can be abused for identity theft: social security numbers, credit card numbers, bank account numbers, health information.

1.4.3 Device Identifier

Device identifiers that may identify a person

1.4.4 Sensitive data

Highly sensitive data whose compromise would lead to reputational harm and/or financial costs.

1.4.5 Data whose protection is a legal obligation

Data whose protection is a legal obligation.

1.4.6 Technical data

Technical data generated by the app (or its related systems) and used to protect other data or the system itself (e.g., encryption keys).

1.4.7 Rulebook

1. *Identify sensitive data according to data classification policies (Required)*

1.4.7.1 Identify sensitive data according to data classification policies (Required)

Identify sensitive data according to a data classification policy that is clearly defined by the industry, country, and organization. If a data classification policy is not available, use the following list of information generally considered confidential.

- User authentication information (credentials, PINs, etc.)
- Personally Identifiable Information (PII) that can be abused for identity theft: social security numbers, credit card numbers, bank account numbers, health information
- Device identifiers that may identify a person
- Highly sensitive data whose compromise would lead to reputational harm and/or financial costs
- Data whose protection is a legal obligation
- Technical data generated by the app (or its related systems) and used to protect other data or the system itself (e.g., encryption keys).

A definition of "sensitive data" must be decided before testing begins because detecting sensitive data leakage without a definition may be impossible.

If this is violated, the following may occur.

- Sensitive data that could be compromised based on the results of penetration testing may not be recognized as sensitive data, and may not be identified and addressed as a risk.

1.5 MSTG-ARCH-12

The app should comply with privacy laws and regulations.

1.5.1 General Privacy Laws and Regulations

Reference

- V6.1 Data Classification

1.5.1.1 Personal Information and Privacy

If personal information is handled, it must comply with GDPR and Act on the Protection of Personal Information.

1.5.1.2 Medical Data

If medical data is handled, it must be HIPAA and HITECH compliant.

1.5.1.3 Financial Information

Credit Card Information

If credit card information is handled, it must be PCI DSS compliant.

1.5.2 Privacy rules on Google Play

We're committed to protecting user privacy and providing a safe and secure environment for our users. Apps that are deceptive, malicious, or intended to abuse or misuse any network, device, or personal data are strictly prohibited.

Reference

- Google Play Policy Center Privacy, Deception and Device Abuse

1.5.2.1 User Data

You must be transparent in how you handle user data (for example, information collected from or about a user, including device information). That means disclosing the access, collection, use, handling, and sharing of user data from your app, and limiting the use of the data to the policy compliant purposes disclosed. Please be aware that any handling of personal and sensitive user data is also subject to additional requirements in the "Personal and Sensitive User Data" section below. These Google Play requirements are in addition to any requirements prescribed by applicable privacy and data protection laws.

If you include third party code (for example, an SDK) in your app, you must ensure that the third party code used in your app, and that third party's practices with respect to user data from your app, are compliant with Google Play Developer Program policies, which include use and disclosure requirements. For example, you must ensure that your SDK providers do not sell personal and sensitive user data from your app. This requirement applies regardless of whether user data is transferred after being sent to a server, or by embedding third-party code in your app.

Personal and Sensitive User Data

Personal and sensitive user data includes, but isn't limited to, personally identifiable information, financial and payment information, authentication information, phonebook, contacts, device location, SMS and call related data, health data, Health Connect data, inventory of other apps on the device, microphone, camera, and other sensitive device or usage data. If your app handles personal and sensitive user data, then you must:

- Limit the access, collection, use, and sharing of personal and sensitive user data acquired through the app to app and service functionality and policy conforming purposes reasonably expected by the user:
  - Apps that extend usage of personal and sensitive user data for serving advertising must comply with Google Play's Ads Policy.
  - You may also transfer data as necessary to service providers or for legal reasons such as to comply with a valid governmental request, applicable law, or as part of a merger or acquisition with legally adequate notice to users.

- Handle all personal and sensitive user data securely, including transmitting it using modern cryptography (for example, over HTTPS).
- Use a runtime permissions request whenever available, prior to accessing data gated by Android permissions.
- Not sell personal and sensitive user data.
  - "Sale" means the exchange or transfer of personal and sensitive user data to a third party for monetary consideration.
    - User-initiated transfer of personal and sensitive user data (for example, when the user is using a feature of the app to transfer a file to a third party, or when the user chooses to use a dedicated purpose research study app), is not regarded as sale.

Prominent Disclosure & Consent Requirement

In cases where your app's access, collection, use, or sharing of personal and sensitive user data may not be within the reasonable expectation of the user of the product or feature in question (for example, if data collection occurs in the background when the user is not engaging with your app), you must meet the following requirements:

Prominent disclosure: You must provide an in-app disclosure of your data access, collection, use, and sharing. The in-app disclosure:

- Must be within the app itself, not only in the app description or on a website;
- Must be displayed in the normal usage of the app and not require the user to navigate into a menu or settings;
- Must describe the data being accessed or collected;
- Must explain how the data will be used and/or shared;
- Cannot only be placed in a privacy policy or terms of service; and
- Cannot be included with other disclosures unrelated to personal and sensitive user data collection.

Consent and runtime permissions: Requests for in-app user consent and runtime permission requests must be immediately preceded by an in-app disclosure that meets the requirement of this policy. The app's request for consent:

- Must present the consent dialog clearly and unambiguously;
- Must require affirmative user action (for example, tap to accept, tick a check-box);
- Must not interpret navigation away from the disclosure (including tapping away or pressing the back or home button) as consent;
- Must not use auto-dismissing or expiring messages as a means of obtaining user consent; and
- Must be granted by the user before your app can begin to collect or access the personal and sensitive user data.

Apps that rely on other legal bases to process personal and sensitive user data without consent, such as a legitimate interest under the EU GDPR, must comply with all applicable legal requirements and provide appropriate disclosures to the users, including in-app disclosures as required under this policy.

To meet policy requirements, it's recommended that you reference the following example format for Prominent Disclosure when it's required:

- "This app" collects/transmits/syncs/stores "type of data" to enable "feature", "in what scenario."
- Example: "Fitness Funds collects location data to enable fitness tracking even when the app is closed or not in use and is also used to support advertising."
- Example: "Call buddy collects read and write call log data to enable contact organization even when the app is not in use."

If your app integrates third party code (for example, an SDK) that is designed to collect personal and sensitive user data by default, you must, within 2 weeks of receipt of a request from Google Play (or, if Google Play's request provides for a longer time period, within that time period), provide sufficient evidence demonstrating that your app meets the Prominent Disclosure and Consent requirements of this policy, including with regard to the data access, collection, use, or sharing via the third party code.

Examples of Common Violations

- An app collects device location but does not have a prominent disclosure explaining which feature uses this data and/or indicates the app's usage in the background.
- An app has a runtime permission requesting access to data before the prominent disclosure which specifies what the data is used for.
- An app that accesses a user's inventory of installed apps and doesn't treat this data as personal or sensitive data subject to the above Privacy Policy, data handling, and Prominent Disclosure and Consent requirements.
- An app that accesses a user's phone or contact book data and doesn't treat this data as personal or sensitive data subject to the above Privacy Policy, data handling, and Prominent Disclosure and Consent requirements.
- An app that records a user's screen and doesn't treat this data as personal or sensitive data subject to this policy.
- An app that collects device location and does not comprehensively disclose its use and obtain consent in accordance with the above requirements.
- An app that uses restricted permissions in the background of the app including for tracking, research, or marketing purposes and does not comprehensively disclose its use and obtain consent in accordance with the above requirements.
- An app with an SDK that collects personal and sensitive user data and doesn't treat this data as subject to this User Data Policy, access, data handling (including disallowed sale), and prominent disclosure and consent requirements.

Refer to this article for more information on the Prominent Disclosure and Consent requirement.

Restrictions for Personal and Sensitive Data Access

In addition to the requirements above, the table below describes requirements for specific activities.

Table 1.5.2.1.1 Requirements for specific operations on user data

Activity	Requirement
Your app handles financial or payment information or government identification numbers	Your app must never publicly disclose any personal and sensitive user data related to financial or payment activities or any government identification numbers.
Your app handles non-public phonebook or contact information	We don't allow unauthorized publishing or disclosure of people's non-public contacts.
Your app contains anti-virus or security functionality, such as anti-virus, anti-malware, or security-related features	Your app must post a privacy policy that, together with any in-app disclosures, explain what user data your app collects and transmits, how it's used, and the type of parties with whom it's shared.
Your app targets children	Your app must not include an SDK that is not approved for use in child-directed services. See Designing Apps for Children and Families for full policy language and requirements.
Your app collects or links persistent device identifiers (e.g., IMEI, IMSI, SIM Serial #, etc.)	Persistent device identifiers may not be linked to other personal and sensitive user data or resettable device identifiers except for the purposes of ・Telephony linked to a SIM identity (e.g., wifi calling linked to a carrier account), and ・Enterprise device management apps using device owner mode. These uses must be prominently disclosed to users as specified in the User Data Policy. Please consult this resource for alternative unique identifiers. Please read the Ads policy for additional guidelines for Android Advertising ID.

Data safety section

All developers must complete a clear and accurate Data safety section for every app detailing collection, use, and sharing of user data. The developer is responsible for the accuracy of the label and keeping this information up-to-date. Where relevant, the section must be consistent with the disclosures made in the app's privacy policy.

Please refer to this article for additional information on completing the Data safety section.

Privacy Policy

All apps must post a privacy policy link in the designated field within Play Console, and a privacy policy link or text within the app itself. The privacy policy must, together with any in-app disclosures, comprehensively disclose how your app accesses, collects, uses, and shares user data, not limited by the data disclosed in the privacy label. This must include:

- Developer information and a privacy point of contact or a mechanism to submit inquiries.
- Disclosing the types of personal and sensitive user data your app accesses, collects, uses, and shares; and any parties with which any personal or sensitive user data is shared.
- Secure data handling procedures for personal and sensitive user data.
- The developer's data retention and deletion policy.
- Clear labeling as a privacy policy (for example, listed as "privacy policy" in title).

The entity (for example, developer, company) named in the app's Google Play store listing must appear in the privacy policy or the app must be named in the privacy policy. Apps that do not access any personal and sensitive user data must still submit a privacy policy.

Please make sure your privacy policy is available on an active, publicly accessible and non-geofenced URL (no PDFs) and is non-editable.

Usage of App Set ID

Android will introduce a new ID to support essential use cases such as analytics and fraud prevention. Terms for the use of this ID are below.

- Usage: App set ID must not be used for ads personalization and ads measurement.
- Association with personally-identifiable information or other identifiers: App set ID may not be connected to any Android identifiers (e.g., AAID) or any personal and sensitive data for advertising purposes.
- Transparency and consent: The collection and use of the app set ID and commitment to these terms must be disclosed to users in a legally adequate privacy notification, including your privacy policy. You must obtain users' legally valid consent where required. To learn more about our privacy standards, please review our User Data policy.

EU-U.S., Swiss Privacy Shield

If you access, use, or process personal information made available by Google that directly or indirectly identifies an individual and that originated in the European Union or Switzerland ("EU Personal Information"), then you must:

- Comply with all applicable privacy, data security, and data protection laws, directives, regulations, and rules;
- Access, use or process EU Personal Information only for purposes that are consistent with the consent obtained from the individual to whom the EU Personal Information relates;
- Implement appropriate organizational and technical measures to protect EU Personal Information against loss, misuse, and unauthorized or unlawful access, disclosure, alteration and destruction; and
- Provide the same level of protection as is required by the Privacy Shield Principles.

You must monitor your compliance with these conditions on a regular basis. If, at any time, you cannot meet these conditions (or if there is a significant risk that you will not be able to meet them), you must immediately notify us by email to data-protection-office@google.com and immediately either stop processing EU Personal Information or take reasonable and appropriate steps to restore an adequate level of protection.

As of July 16, 2020, Google no longer relies on the EU-U.S. Privacy Shield to transfer personal data that originated in the European Economic Area or the UK to the United States. (Learn More.) More information is set forth in Section 9 of the DDA.

Reference

- Google Play Developer Policy Center User Data

1.5.2.2 Permissions and APIs that Access Sensitive Information

Requests for permission and APIs that access sensitive information should make sense to users. You may only request permissions and APIs that access sensitive information that are necessary to implement current features or services in your app that are promoted in your Google Play listing. You may not use permissions or APIs that access sensitive information that give access to user or device data for undisclosed, unimplemented, or disallowed features or purposes. Personal or sensitive data accessed through permissions or APIs that access sensitive information may never be sold nor shared for a purpose facilitating sale.

Request permissions and APIs that access sensitive information to access data in context (via incremental requests), so that users understand why your app is requesting the permission. Use the data only for purposes that the user has consented to. If you later wish to use the data for other purposes, you must ask users and make sure they affirmatively agree to the additional uses.

Restricted Permissions

In addition to the above, restricted permissions are permissions that are designated as Dangerous, Special, Signature, or as documented below. These permissions are subject to the following additional requirements and restrictions:

- User or device data accessed through Restricted Permissions is considered as personal and sensitive user data. The requirements of the User Data policy apply.
- Respect users' decisions if they decline a request for a Restricted Permission, and users may not be manipulated or forced into consenting to any non-critical permission. You must make a reasonable effort to accommodate users who do not grant access to sensitive permissions (for example, allowing a user to manually enter a phone number if they' ve restricted access to Call Logs).
- Use of permissions in violation of Google Play malware policies (including Elevated Privilege Abuse) is expressly prohibited.

Certain Restricted Permissions may be subject to additional requirements as detailed below. The objective of these restrictions is to safeguard user privacy. We may make limited exceptions to the requirements below in very rare cases where apps provide a highly compelling or critical feature and where there is no alternative method available to provide the feature. We evaluate proposed exceptions against the potential privacy or security impacts on users.

SMS and Call Log Permissions

SMS and Call Log Permissions are regarded as personal and sensitive user data subject to the Personal and Sensitive Information policy, and the following restrictions:

Table 1.5.2.2.1 Limited Permissions and Requirements

Restricted Permission	Requirement
Call Log permission group (e.g. READ_CALL_LOG, WRITE_CALL_LOG, PROCESS_OUTGOING_CALLS)	It must be actively registered as the default Phone or Assistant handler on the device.
SMS permission group (e.g. READ_SMS, SEND_SMS, WRITE_SMS, RECEIVE_SMS, RECEIVE_WAP_PUSH, RECEIVE_MMS)	It must be actively registered as the default SMS or Assistant handler on the device.

Apps lacking default SMS, Phone, or Assistant handler capability may not declare use of the above permissions in the manifest. This includes placeholder text in the manifest. Additionally, apps must be actively registered as the default SMS, Phone, or Assistant handler before prompting users to accept any of the above permissions and must immediately stop using the permission when they' re no longer the default handler. The permitted uses and exceptions are available on this Help Center page.

Apps may only use the permission (and any data derived from the permission) to provide approved core app functionality Core functionality is defined as the main purpose of the app. This may include a set of core features, which must all be prominently documented and promoted in the app' s description. Without the core feature(s), the app is "broken" or rendered unusable. The transfer, sharing, or licensed use of this data must only be for providing core features or services within the app, and its use may not be extended for any other purpose (e.g., improving other apps or services, advertising, or marketing purposes). You may not use alternative methods (including other permissions, APIs, or third-party sources) to derive data attributed to Call Log or SMS related permissions.

Location Permissions

Device location is regarded as personal and sensitive user data subject to the Personal and Sensitive Information policy and the Background Location policy, and the following requirements:

- Apps may not access data protected by location permissions (e.g., ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, ACCESS_BACKGROUND_LOCATION) after it is no longer necessary to deliver current features or services in your app.
- You should never request location permissions from users for the sole purpose of advertising or analytics. Apps that extend permitted usage of this data for serving advertising must be in compliance with our Ads Policy.
- Apps should request the minimum scope necessary (i.e., coarse instead of fine, and foreground instead of background) to provide the current feature or service requiring location and users should reasonably expect that the feature or service needs the level of location requested. For example, we may reject apps that request or access background location without compelling justification. Background location may only be used to provide features beneficial to the user and relevant to the core functionality of the app.

Apps are allowed to access location using foreground service (when the app only has foreground access e.g., "while in use") permission if the use:

- has been initiated as a continuation of an in-app user-initiated action, and
- is terminated immediately after the intended use case of the user-initiated action is completed by the application.

Apps designed specifically for children must comply with the Designed for Families policy.

For more information on the policy requirements, please see this help article.

All Files Access Permission

Files and directory attributes on a user' s device are regarded as personal and sensitive user data subject to the Personal and Sensitive Information policy and the following requirements:

- Apps should only request access to device storage which is critical for the app to function, and may not request access to device storage on behalf of any third-party for any purpose that is unrelated to critical user-facing app functionality.
- Android devices running R or later, will require the MANAGE_EXTERNAL_STORAGE permission in order to manage access in shared storage. All apps that target R and request broad access to shared storage ("All files access") must successfully pass an appropriate access review prior to publishing. Apps allowed to use this permission must clearly prompt users to enable "All files access" for their app under "Special app access" settings. For more information on the R requirements, please see this help article.

Package (App) Visibility Permission

The inventory of installed apps queried from a device are regarded as personal and sensitive user data subject to the Personal and Sensitive Information policy, and the following requirements:

Apps that have a core purpose to launch, search, or interoperate with other apps on the device, may obtain scope-appropriate visibility to other installed apps on the device as outlined below:

- **Broad app visibility**: Broad visibility is the capability of an app to have extensive (or "broad") visibility of the installed apps ("packages") on a device.
  - For apps targeting API level 30 or later, broad visibility to installed apps via the QUERY_ALL_PACKAGES permission is restricted to specific use cases where awareness of and/or interoperability with any and all apps on the device are required for the app to function.

        * You may not use QUERY_ALL_PACKAGES if your app can operate with a more targeted scoped package visibility declaration(e.g. querying and interacting with specific packages instead of requesting broad visibility).
    - Use of alternative methods to approximate the broad visibility level associated with QUERY_ALL_PACKAGES permission are also restricted to user-facing core app functionality and interoperability with any apps discovered via this method.
    - Please see this Help Center article for allowable use cases for the QUERY_ALL_PACKAGES permission.
- **Limited app visibility**: Limited visibility is when an app minimizes access to data by querying for specific apps using more targeted (instead of "broad") methods (e.g. querying for specific apps that satisfy your app' s manifest declaration). You may use this method to query for apps in cases where your app has policy compliant interoperability, or management of these apps.
- Visibility to the inventory of installed apps on a device must be directly related to the core purpose or core functionality that users access within your app.

App inventory data queried from Play-distributed apps may never be sold nor shared for analytics or ads monetization purposes.

Accessibility API

The Accessibility API cannot be used to:

- Change user settings without their permission or prevent the ability for users to disable or uninstall any app or service unless authorized by a parent or guardian through a parental control app or by authorized administrators through enterprise management software;
- Work around Android built-in privacy controls and notifications; or
- Change or leverage the user interface in a way that is deceptive or otherwise violates Google Play Developer Policies.

The Accessibility API is not designed and cannot be requested for remote call audio recording.

The use of the Accessibility API must be documented in the Google Play listing.

Guidelines for IsAccessibilityTool

Apps with a core functionality intended to directly support people with disabilities are eligible to use the **IsAccessibilityTool** to appropriately publicly designate themselves as an accessibility app.

Apps not eligible for **IsAccessibilityTool** may not use the flag and must meet prominent disclosure and consent requirements as outlined in the User Data policy as the accessibility related functionality is not obvious to the user. Please refer to the AccessibilityService API help center article for more information.

Apps must use more narrowly scoped APIs and permissions in lieu of the Accessibility API when possible to achieve the desired functionality.

Request Install Packages Permission

The REQUEST_INSTALL_PACKAGES permission allows an application to request the installation of app packages. To use this permission, your app' s core functionality must include:

- Sending or receiving app packages; and
- Enabling user-initiated installation of app packages.

Permitted functionalities include:

- Web browsing or search; or
- Communication services that support attachments; or
- File sharing, transfer or management; or
- Enterprise device management.
- Backup & restore

- Device Migration / Phone Transfer

Core functionality is defined as the main purpose of the app. The core functionality, as well as any core features that comprise this core functionality, must all be prominently documented and promoted in the app's description.

The REQUEST_INSTALL_PACKAGES permission may not be used to perform self updates, modifications, or the bundling of other APKs in the asset file unless for device management purposes. All updates or installing of packages must abide by Google Play's Device and Network Abuse policy and must be initiated and driven by the user.

Health Connect by Android Permissions

Data accessed through Health Connect Permissions is regarded as personal and sensitive user data subject to the User Data policy, and the following additional requirements:

Appropriate Access to and Use of Health Connect

Requests to access data through Health Connect must be clear and understandable. Health Connect may only be used in accordance with the applicable policies, terms and conditions, and for approved use cases as set forth in this policy. This means you may only request access to permissions when your application or service meets one of the approved use cases.

Approved use cases for access to Health Connect Permissions are:

- Applications or services with one or more features to benefit users' health and fitness via a user interface allowing users to directly **journal, report, monitor, and/or analyze** their physical activity, sleep, mental well-being, nutrition, health measurements, physical descriptions, and/or other health or fitness-related descriptions and measurements.
- Applications or services with one or more features to benefit users' health and fitness via a user interface allowing users to **store** their physical activity, sleep, mental well-being, nutrition, health measurements, physical descriptions, and/or other health or fitness-related descriptions and measurements on their phone and/or wearable, and share their data with other on-device apps that satisfy these use cases.

Health Connect is a general purpose data storage and sharing platform that allows users to aggregate health and fitness data from various sources on their Android device and share it with third parties at their election. The data may originate from various sources as determined by the users. Developers must assess whether Health Connect is appropriate for their intended use and to investigate and vet the source and quality of any data from Health Connect in connection with any purpose, and, in particular, for research, health, or medical uses.

- Apps conducting health-related human subject research using data obtained through Health Connect must obtain consent from participants or, in the case of minors, their parent or guardian. Such consent must include the (a) nature, purpose, and duration of the research; (b) procedures, risks, and benefits to the participant; (c) information about confidentiality and handling of data (including any sharing with third parties); (d) a point of contact for participant questions; and (e) the withdrawal process. Apps conducting health-related human subject research using data obtained through Health Connect must receive approval from an independent board whose aim is 1) to protect the rights, safety, and well-being of participants and 2) with the authority to scrutinize, modify, and approve human subjects research. Proof of such approval must be provided upon request.
- It is also your responsibility for ensuring compliance with any regulatory or legal requirements that may apply based on your intended use of Health Connect and any data from Health Connect. Except as explicitly noted in the labeling or information provided by Google for specific Google products or services, Google does not endorse the use of or warrant the accuracy of any data contained in Health Connect for any use or purpose, and, in particular, for research, health, or medical uses. Google disclaims all liability associated with use of data obtained through Health Connect.

Limited Use

Upon using Health Connect for an appropriate use, your use of the data accessed through Health Connect must also comply with the below requirements. These requirements apply to the raw data obtained from Health Connect, and data aggregated, de-identified, or derived from the raw data.

- Limit your use of Health Connect data to providing or improving your appropriate use case or features that are visible and prominent in the requesting application's user interface.
- Only transfer user data to third parties:

  - To provide or improve your appropriate use case or features that are clear from the requesting application's user interface and only with the user's consent;
  - If necessary for security purposes (for example, investigating abuse);
  - To comply with applicable laws and/or regulations; or,
  - As part of a merger, acquisition or sale of assets of the developer after obtaining explicit prior consent from the user.
- Do not allow humans to read user data, unless:
  - The user's explicit consent to read specific data is obtained;
  - It's necessary for security purposes (for example, investigating abuse);
  - To comply with applicable laws; or,
  - The data (including derivations) is aggregated and used for internal operations in accordance with applicable privacy and other jurisdictional legal requirements.

All other transfers, uses, or sale of Health Connect data is prohibited, including:

- Transferring or selling user data to third parties like advertising platforms, data brokers, or any information resellers.
- Transferring, selling, or using user data for serving ads, including personalized or interest-based advertising.
- Transferring, selling, or using user data to determine credit-worthiness or for lending purposes.
- Transferring, selling, or using the user data with any product or service that may qualify as a medical device pursuant to Section 201(h) of the Federal Food Drug & Cosmetic Act if the user data will be used by the medical device to perform its regulated function.
- Transferring, selling, or using user data for any purpose or in any manner involving Protected Health Information (as defined by HIPAA) unless you receive prior written approval to such use from Google.

Access to Health Connect may not be used in violation of this policy or other applicable Health Connect terms and conditions or policies, including for the following purposes:

- Do not use Health Connect in developing, or for incorporation into, applications, environments or activities where the use or failure of Health Connect could reasonably be expected to lead to death, personal injury, or environmental or property damage (such as the creation or operation of nuclear facilities, air traffic control, life support systems, or weaponry).
- Do not access data obtained through Health Connect using headless apps. Apps must display a clearly identifiable icon in the app tray, device app settings, notification icons, etc.
- Do not use Health Connect with apps that sync data between incompatible devices or platforms.
- Health Connect cannot connect to applications, services or features that solely target children. Health Connect is not approved for use in primarily child-directed services.

An affirmative statement that your use of Health Connect data complies with Limited Use restrictions must be disclosed in your application or on a website belonging to your web-service or application; for example, a link on a homepage to a dedicated page or privacy policy noting: "The use of information received from Health Connect will adhere to the Health Connect Permissions policy, including the Limited Use requirements."

Minimum Scope

You may only request access to permissions that are critical to implementing your application or service's functionality.

This means:

- Don't request access to information that you don't need. Only request access to the permissions necessary to implement your product's features or services. If your product does not require access to specific permissions, then you must not request access to these permissions.

Transparent and Accurate Notice and Control

Health Connect handles health and fitness data, which includes personal and sensitive information. All applications and services must contain a privacy policy, which must comprehensively disclose how your application or service collects, uses, and shares user data. This includes the types of parties to which any user data is shared, how you use the data, how you store and secure the data, and what happens to the data when an account is deactivated and/or deleted.

In addition to the requirements under applicable law, you must also adhere to the following requirements:

- You must provide a disclosure of your data access, collection, use, and sharing. The disclosure:
  - Must accurately represent the identity of the application or service that seeks access to user data;
  - Must provide clear and accurate information explaining the types of data being accessed, requested, and/or collected;
  - Must explain how the data will be used and/or shared: if you request data for one reason, but the data will also be utilized for a secondary purpose, you must notify users of both use cases.
- You must provide user help documentation that explains how users can manage and delete their data from your app.

Secure Data Handling

You must handle all user data securely. Take reasonable and appropriate steps to protect all applications or systems that make use of Health Connect against unauthorized or unlawful access, use, destruction, loss, alteration, or disclosure.

Recommended security practices include implementing and maintaining an Information Security Management System such as outlined in ISO/IEC 27001 and ensuring your application or web service is robust and free from common security issues as set out by the OWASP Top 10.

Depending on the API being accessed and number of user grants or users, we will require that your application or service undergo a periodic security assessment and obtain a Letter of Assessment from a designated third party if your product transfers data off the user' s own device.

For more information on requirements for apps connecting to Health Connect, please see this help article.

VPN Service

The VpnService is a base class for applications to extend and build their own VPN solutions. Only apps that use the VpnService and have VPN as their core functionality can create a secure device-level tunnel to a remote server. Exceptions include apps that require a remote server for core functionality such as:

- Parental control and enterprise management apps.
- App usage tracking.
- Device security apps (for example, anti-virus, mobile device management, firewall).
- Network related tools (for example, remote access).
- Web browsing apps.
- Carrier apps that require the use of VPN functionality to provide telephony or connectivity services.

The VpnService cannot be used to:

- Collect personal and sensitive user data without prominent disclosure and consent.
- Redirect or manipulate user traffic from other apps on a device for monetization purposes (for example, redirecting ads traffic through a country different than that of the user).
- Manipulate ads that can impact apps monetization.

Apps that use the VpnService must:

- Document use of the VpnService in the Google Play listing, and
- Must encrypt the data from the device to VPN tunnel end point, and
- Abide by all Developer Program Policies including the Ad Fraud, Permissions, and Malware policies.

Reference

- Google Play Developer Policy Center Permissions and APIs that Access Sensitive Information

1.5.2.3 Device and Network Abuse

We don't allow apps that interfere with, disrupt, damage, or access in an unauthorized manner the user's device, other devices or computers, servers, networks, application programming interfaces (APIs), or services, including but not limited to other apps on the device, any Google service, or an authorized carrier's network.

Apps on Google Play must comply with the default Android system optimization requirements documented in the Core App Quality guidelines for Google Play.

An app distributed via Google Play may not modify, replace, or update itself using any method other than Google Play's update mechanism. Likewise, an app may not download executable code (e.g., dex, JAR, .so files) from a source other than Google Play. This restriction does not apply to code that runs in a virtual machine or an interpreter where either provides indirect access to Android APIs (such as JavaScript in a webview or browser).

Apps or third-party code (e.g., SDKs) with interpreted languages (JavaScript, Python, Lua, etc.) loaded at run time (e.g., not packaged with the app) must not allow potential violations of Google Play policies.

We don't allow code that introduces or exploits security vulnerabilities. Check out the App Security Improvement Program to find out about the most recent security issues flagged to developers.

Flag Secure Requirements

FLAG_SECURE is a display flag declared in an app's code to indicate that its UI contains sensitive data intended to be limited to a secure surface while using the app. This flag is designed to prevent the data from appearing in screenshots or from being viewed on non-secure displays. Developers declare this flag when the app's content should not be broadcast, viewed, or otherwise transmitted outside of the app or users' device.

For security and privacy purposes, all apps distributed on Google Play are required to respect the FLAG_SECURE declaration of other apps. Meaning, apps must not facilitate or create workarounds to bypass the FLAG_SECURE settings in other apps.

Apps that qualify as an Accessibility Tool are exempt from this requirement, as long as they do not transmit, save, or cache FLAG_SECURE protected content for access outside of the user's device.

Examples of Common Violations

- Apps that block or interfere with another app displaying ads.
- Game cheating apps that affect the gameplay of other apps.
- Apps that facilitate or provide instructions on how to hack services, software or hardware, or circumvent security protections.
- Apps that access or use a service or API in a manner that violates its terms of service.
- Apps that are not eligible for whitelisting and attempt to bypass system power management .
- Apps that facilitate proxy services to third parties may only do so in apps where that is the primary, user-facing core purpose of the app.
- Apps or third party code (e.g., SDKs) that download executable code, such as dex files or native code, from a source other than Google Play.
- Apps that install other apps on a device without the user's prior consent.
- Apps that link to or facilitate the distribution or installation of malicious software.
- Apps or third party code (e.g., SDKs) containing a webview with added JavaScript Interface that loads untrusted web content (e.g., http:// URL) or unverified URLs obtained from untrusted sources (e.g., URLs obtained with untrusted Intents).

Reference

- Google Play Developer Policy Center Device and Network Abuse

1.5.2.4 Deceptive Behavior

We don't allow apps that attempt to deceive users or enable dishonest behavior including but not limited to apps which are determined to be functionally impossible. Apps must provide an accurate disclosure, description and images/video of their functionality in all parts of the metadata. Apps must not attempt to mimic functionality or warnings from the operating system or other apps. Any changes to device settings must be made with the user's knowledge and consent and be reversible by the user.

Misleading Claims

We don't allow apps that contain false or misleading information or claims, including in the description, title, icon, and screenshots.

Examples of Common Violations

- Apps that misrepresent or do not accurately and clearly describe their functionality:
  - An app that claims to be a racing game in its description and screenshots, but is actually a puzzle block game using a picture of a car.
  - An app that claims to be an antivirus app, but only contains a text guide explaining how to remove viruses.
- Apps that claim functionalities that are not possible to implement, such as insect repellent apps, even if it is represented as a prank, fake, joke, etc.
- Apps that are improperly categorized, including but not limited to the app rating or app category.
- Demonstrably deceptive or false content that may interfere with voting processes.
- Apps that falsely claim affiliation with a government entity or to provide or facilitate government services for which they are not properly authorized.
- Apps that falsely claim to be the official app of an established entity. Titles like "Justin Bieber Official" are not allowed without the necessary permissions or rights.

Deceptive Device Settings Changes

We don't allow apps that make changes to the user's device settings or features outside of the app without the user's knowledge and consent. Device settings and features include system and browser settings, bookmarks, shortcuts, icons, widgets, and the presentation of apps on the homescreen.

Additionally, we do not allow:

- Apps that modify device settings or features with the user's consent but do so in a way that is not easily reversible.
- Apps or ads that modify device settings or features as a service to third parties or for advertising purposes.
- Apps that mislead users into removing or disabling third-party apps or modifying device settings or features.
- Apps that encourage or incentivize users into removing or disabling third-party apps or modifying device settings or features unless it is part of a verifiable security service.

Enabling Dishonest Behavior

We don't allow apps that help users to mislead others or are functionally deceptive in any way, including, but not limited to: apps that generate or facilitate the generation of ID cards, social security numbers, passports, diplomas, credit cards, bank accounts, and driver's licenses. Apps must provide accurate disclosures, titles, descriptions, and images/video regarding the app's functionality and/or content and should perform as reasonably and accurately expected by the user.

Additional app resources (for example, game assets) may only be downloaded if they are necessary for the users' use of the app. Downloaded resources must be compliant with all Google Play policies, and before beginning the download, the app should prompt users and clearly disclose the download size.

Any claim that an app is a "prank", "for entertainment purposes" (or other synonym) does not exempt an app from application of our policies.

Examples of Common Violations

- Apps that mimic other apps or websites to trick users into disclosing personal or authentication information.
- Apps that depict or display unverified or real world phone numbers, contacts, addresses, or personally identifiable information of non-consenting individuals or entities.
- Apps with different core functionality based on a user's geography, device parameters, or other user-dependent data where those differences are not prominently advertised to the user in the store listing.
- Apps that change significantly between versions without alerting the user (e.g., 'what's new' section) and updating the store listing.
- Apps that attempt to modify or obfuscate behavior during review.
- Apps with content delivery network (CDN) facilitated downloads that fail to prompt the user and disclose the download size prior to downloading.

Manipulated Media

We don't allow apps that promote or help create false or misleading information or claims conveyed through imagery, videos and/or text. We disallow apps determined to promote or perpetuate demonstrably misleading or deceptive imagery, videos and/or text, which may cause harm pertaining to a sensitive event, politics, social issues, or other matters of public concern.

Apps that manipulate or alter media, beyond conventional and editorially acceptable adjustments for clarity or quality, must prominently disclose or watermark altered media when it may not be clear to the average person that the media has been altered. Exceptions may be provided for public interest or obvious satire or parody.

Examples of Common Violations

- Apps adding a public figure to a demonstration during a politically sensitive event.
- Apps using public figures or media from a sensitive event to advertise media altering capability within an app's store listing.
- Apps that alter media clips to mimic a news broadcast.

Reference

- Google Play Developer Policy Center Deceptive Behavior

1.5.2.5 Misrepresentation

We do not allow apps or developer accounts that:

- impersonate any person or organization, or that misrepresent or conceal their ownership or primary purpose.
- that engage in coordinated activity to mislead users. This includes, but isn't limited to, apps or developer accounts that misrepresent or conceal their country of origin and that direct content at users in another country.
- coordinate with other apps, sites, developers, or other accounts to conceal or misrepresent developer or app identity or other material details, where app content relates to politics, social issues or matters of public concern.

Reference

- Google Play Developer Policy Center Misrepresentation

1.5.2.6 Google Play's Target API Level Policy

To provide users with a safe and secure experience, Google Play requires the following target API levels for **all apps**:

New apps and app updates MUST target an Android API level within one year of the latest major Android version release. New apps and app updates that fail to meet this requirement will be prevented from app submission in Play Console.

Existing Google Play apps that are not updated and that do not target an API level within two years of the latest major Android version release, will not be available to new users with devices running newer versions of Android OS. Users who have previously installed the app from Google Play will continue to be able to discover, re-install, and use the app on any Android OS version that the app supports.

For technical advice on how to meet the target API level requirement, please consult the migration guide.

For exact timelines, please refer to this Help Center article.

Reference

- Google Play Developer Policy Center Google Play's Target API Level Policy

1.5.3 Intellectual Property Rights/Intellectual Property Rules on Google Play

We don't allow apps or developer accounts that infringe on the intellectual property rights of others (including trademark, copyright, patent, trade secret, and other proprietary rights). We also don't allow apps that encourage or induce infringement of intellectual property rights.

We will respond to clear notices of alleged copyright infringement. For more information or to file a DMCA request, please visit our copyright procedures.

To submit a complaint regarding the sale or promotion for sale of counterfeit goods within an app, please submit a counterfeit notice.

If you are a trademark owner and you believe there is an app on Google Play that infringes on your trademark rights, we encourage you to reach out to the developer directly to resolve your concern. If you are unable to reach a resolution with the developer, please submit a trademark complaint through this form.

If you have written documentation proving that you have permission to use a third party's intellectual property in your app or store listing (such as brand names, logos and graphic assets), contact the Google Play team in advance of your submission to ensure that your app is not rejected for an intellectual property violation.

Reference

- Google Play Developer Policy Center Intellectual Property

1.5.3.1 Unauthorized Use of Copyrighted Content

We don't allow apps that infringe copyright. Modifying copyrighted content may still lead to a violation. Developers may be required to provide evidence of their rights to use copyrighted content.

Please be careful when using copyrighted content to demonstrate the functionality of your app. In general, the safest approach is to create something that's original.

Examples of Common Violations

- Cover art for music albums, video games, and books.
- Marketing images from movies, television, or video games.
- Artwork or images from comic books, cartoons, movies, music videos, or television.
- College and professional sports team logos.
- Photos taken from a public figure's social media account.
- Professional images of public figures.
- Reproductions or "fan art" indistinguishable from the original work under copyright.

- Apps that have soundboards that play audio clips from copyrighted content.
- Full reproductions or translations of books that are not in the public domain.

1.5.3.2 Encouraging Infringement of Copyright

We don't allow apps that induce or encourage copyright infringement. Before you publish your app, look for ways your app may be encouraging copyright infringement and get legal advice if necessary.

Examples of Common Violations

- Streaming apps that allow users to download a local copy of copyrighted content without authorization.
- Apps that encourage users to stream and download copyrighted works, including music and video, in violation of applicable copyright law:

1.5.3.3 Trademark Infringement

We don't allow apps that infringe on others' trademarks. A trademark is a word, symbol, or combination that identifies the source of a good or service. Once acquired, a trademark gives the owner exclusive rights to the trademark usage with respect to certain goods or services.

Trademark infringement is improper or unauthorized use of an identical or similar trademark in a way that is likely to cause confusion as to the source of that product. If your app uses another party's trademarks in a way that is likely to cause confusion, your app may be suspended.

1.5.3.4 Counterfeit

We don't allow apps that sell or promote for sale counterfeit goods. Counterfeit goods contain a trademark or logo that is identical to or substantially indistinguishable from the trademark of another. They mimic the brand features of the product in an attempt to pass themselves off as a genuine product of the brand owner.

2 Data Storage and Privacy Requirements

2.1 MSTG-STORAGE-1

System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys.

2.1.1 Hardware-backed Android KeyStore

As mentioned before, hardware-backed Android KeyStore gives another layer to defense-in-depth security concept for Android. Keymaster Hardware Abstraction Layer (HAL) was introduced with Android 6 (API level 23). Applications can verify if the key is stored inside the security hardware (by checking if KeyInfo.isinsideSecureHardware returns true). Devices running Android 9 (API level 28) and higher can have a StrongBox Keymaster module, an implementation of the Keymaster HAL that resides in a hardware security module which has its own CPU, Secure storage, a true random number generator and a mechanism to resist package tampering. To use this feature, true must be passed to the setIsStrongBoxBacked method in either the KeyGenParameterSpec.Builder class or the KeyProtection.Builder class when generating or importing keys using AndroidKeystore. To make sure that StrongBox is used during runtime, check that isInsideSecureHardware returns true and that the system does not throw StrongBoxUnavailableException which gets thrown if the StrongBox Keymaster isn't available for the given algorithm and key size associated with a key. Description of features on hardware-based keystore can be found on AOSP pages.

Keymaster HAL is an interface to hardware-backed components - Trusted Execution Environment (TEE) or a Secure Element (SE), which is used by Android Keystore. An example of such a hardware-backed component is Titan M.

Reference

- owasp-mastg Data Storage Methods Overview Hardware-backed Android KeyStore

Rulebook

- *Verify that keys are stored inside security hardware (Recommended)*
- *How to use StrongBox (Recommended)*

2.1.2 Key Attestation

For the applications which heavily rely on Android Keystore for business-critical operations such as multi-factor authentication through cryptographic primitives, secure storage of sensitive data at the client-side, etc. Android provides the feature of Key Attestation which helps to analyze the security of cryptographic material managed through Android Keystore. From Android 8.0 (API level 26), the key attestation was made mandatory for all new (Android 7.0 or higher) devices that need to have device certification for Google apps. Such devices use attestation keys signed by the Google hardware attestation root certificate and the same can be verified through the key attestation process.

During key attestation, we can specify the alias of a key pair and in return, get a certificate chain, which we can use to verify the properties of that key pair. If the root certificate of the chain is the Google Hardware Attestation Root certificate and the checks related to key pair storage in hardware are made it gives an assurance that the device supports hardware-level key attestation and the key is in the hardware-backed keystore that Google believes to be secure. Alternatively, if the attestation chain has any other root certificate, then Google does not make any claims about the security of the hardware.

Although the key attestation process can be implemented within the application directly but it is recommended that it should be implemented at the server-side for security reasons. The following are the high-level guidelines for the secure implementation of Key Attestation:

- The server should initiate the key attestation process by creating a random number securely using CSPRNG(Cryptographically Secure Random Number Generator) and the same should be sent to the user as a challenge.
- The client should call the setAttestationChallenge API with the challenge received from the server and should then retrieve the attestation certificate chain using the KeyStore.getCertificateChain method.
- The attestation response should be sent to the server for the verification and following checks should be performed for the verification of the key attestation response:
  - Verify the certificate chain, up to the root and perform certificate sanity checks such as validity, integrity and trustworthiness. Check the Certificate Revocation Status List maintained by Google, if none of the certificates in the chain was revoked.
  - Check if the root certificate is signed with the Google attestation root key which makes the attestation process trustworthy.
  - Extract the attestation certificate extension data, which appears within the first element of the certificate chain and perform the following checks:
    - Verify that the attestation challenge is having the same value which was generated at the server while initiating the attestation process.
    - Verify the signature in the key attestation response.
    - Verify the security level of the Keymaster to determine if the device has secure key storage mechanism. Keymaster is a piece of software that runs in the security context and provides all the secure keystore operations. The security level will be one of Software, TrustedEnvironment or StrongBox. The client supports hardware-level key attestation if security level is TrustedEnvironment or StrongBox and attestation certificate chain contains a root certificate signed with Google attestation root key.
    - Verify client' s status to ensure full chain of trust - verified boot key, locked bootloader and verified boot state.
    - Additionally, you can verify the key pair' s attributes such as purpose, access time, authentication requirement, etc.

Note, if for any reason that process fails, it means that the key is not in security hardware. That does not mean that the key is compromised.

The typical example of Android Keystore attestation response looks like this:

```
{
    "fmt": "android-key",
    "authData": "9569088f1ecee3232954035dbd10d7cae391305a2751b559bb8fd7cbb229bd...
↪",
    "attStmt": {
        "alg": -7,
        "sig": "304402202ca7a8cfb6299c4a073e7e022c57082a46c657e9e53...",
        "x5c": [

↪"308202ca30820270a003020102020101300a06082a8648ce3d040302308188310b3009060355040613 0.
↪..",

↪"308202783082021ea003020102020210013 00a06082a8648ce3d040302308198310b300906035504061.
↪..",

↪"3082028b30820232a003020102020900a2059ed10e435b57300a06082a8648ce3d040302308198310b3.
↪.."
        ]
    }
}
```

In the above JSON snippet, the keys have the following meaning:

- fmt: Attestation statement format identifier
- authData: It denotes the authenticator data for the attestation
- alg: The algorithm that is used for the Signature
- sig: Signature
- x5c: Attestation certificate chain

Note: The sig is generated by concatenating authData and clientDataHash (challenge sent by the server) and signing through the credential private key using the alg signing algorithm and the same is verified at the server-side by using the public key in the first certificate.

For more understanding on the implementation guidelines, Google Sample Code can be referred.

For the security analysis perspective the analysts may perform the following checks for the secure implementation of Key Attestation:

- Check if the key attestation is totally implemented at the client-side. In such scenario, the same can be easily bypassed by tampering the application, method hooking, etc.
- Check if the server uses random challenge while initiating the key attestation. As failing to do that would lead to insecure implementation thus making it vulnerable to replay attacks. Also, checks pertaining to the randomness of the challenge should be performed.
- Check if the server verifies the integrity of key attestation response.
- Check if the server performs basic checks such as integrity verification, trust verification, validity, etc. on the certificates in the chain.

Reference

- owasp-mastg Data Storage Methods Overview Key Attestation

Rulebook

- *For secure key authentication, provide a certificate in the device by means of a challenge received from the server (Recommended)*
- *Check for secure implementation of key authentication in terms of security analysis (Required)*

2.1.3 Secure Key Import into Keystore

Android 9 (API level 28) adds the ability to import keys securely into the AndroidKeystore. First AndroidKeystore generates a key pair using PURPOSE_WRAP_KEY which should also be protected with an attestation certificate, this pair aims to protect the Keys being imported to AndroidKeystore. The encrypted keys are generated as ASN.1-encoded message in the SecureKeyWrapper format which also contains a description of the ways the imported key is allowed to be used. The keys are then decrypted inside the AndroidKeystore hardware belonging to the specific device that generated the wrapping key so they never appear as plaintext in the device' s host memory.

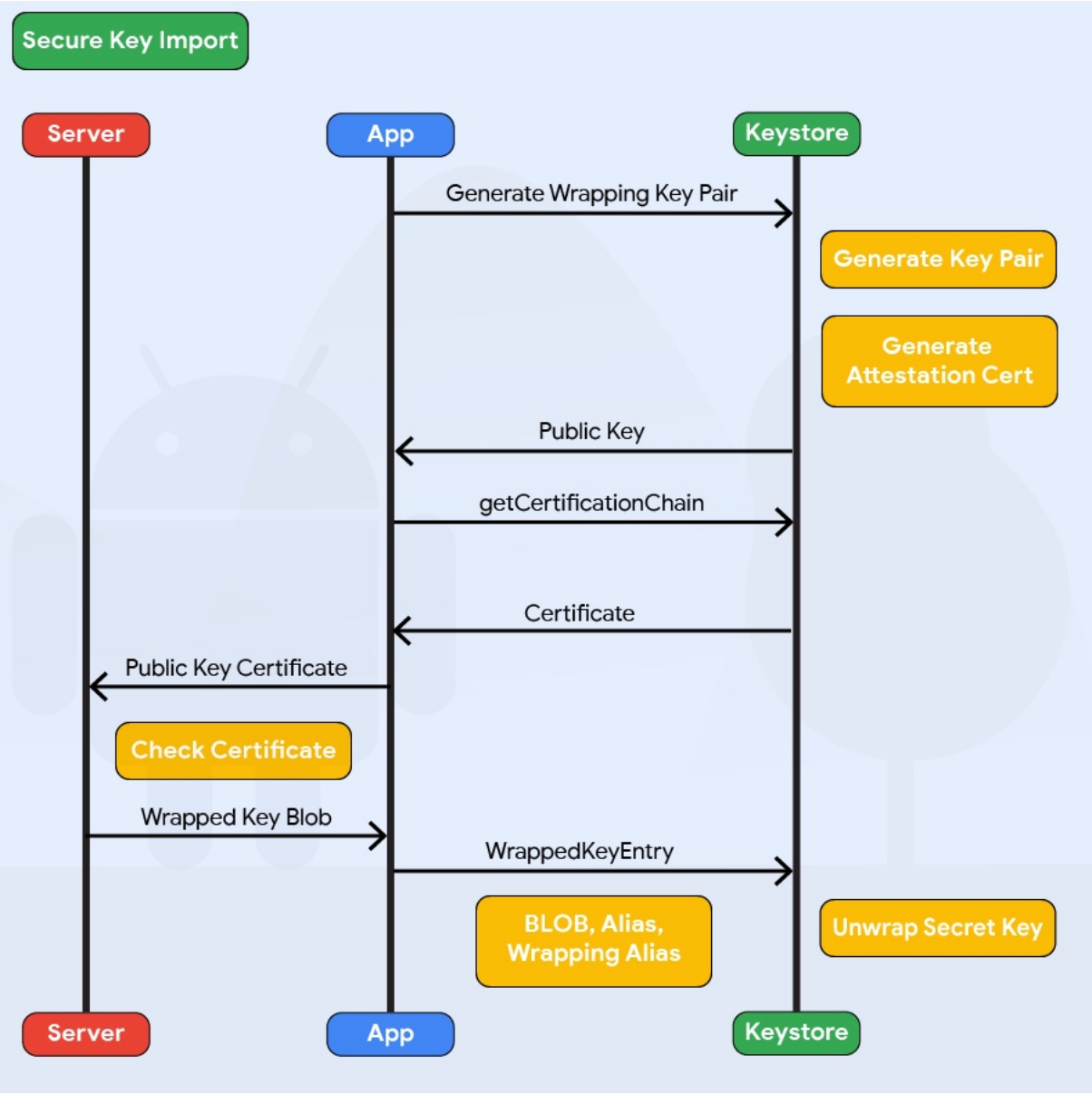


Fig 2.1.3.1 Secure Key Import Flow

Example in Java:

```
KeyDescription ::= SEQUENCE {
    keyFormat INTEGER,
    authorizationList AuthorizationList
}
```

(continues on next page)

(continued from previous page)

```
SecureKeyWrapper ::= SEQUENCE {
    wrapperFormatVersion INTEGER,
    encryptedTransportKey OCTET_STRING,
    initializationVector OCTET_STRING,
    keyDescription KeyDescription,
    secureKey OCTET_STRING,
    tag OCTET_STRING
}
```

The code above present the different parameters to be set when generating the encrypted keys in the SecureKeyWrapper format. Check the Android documentation on WrappedKeyEntry for more details.

When defining the KeyDescription AuthorizationList, the following parameters will affect the encrypted keys security:

- The algorithm parameter Specifies the cryptographic algorithm with which the key is used
- The keySize parameter Specifies the size, in bits, of the key, measuring in the normal way for the key' s algorithm
- The digest parameter Specifies the digest algorithms that may be used with the key to perform signing and verification operations

Reference

- owasp-mastg Data Storage Methods Overview Secure Key Import into Keystore

2.1.4 Older KeyStore Implementations

Older Android versions don' t include KeyStore, but they do include the KeyStore interface from JCA (Java Cryptography Architecture). You can use KeyStores that implement this interface to ensure the secrecy and integrity of keys stored with KeyStore; BouncyCastle KeyStore (BKS) is recommended. All implementations are based on the fact that files are stored on the filesystem; all files are password-protected. To create one, you can use the KeyStore.getInstance("BKS" , "BC") method, where "BKS" is the KeyStore name (BouncyCastle Keystore) and "BC" is the provider (BouncyCastle). You can also use SpongyCastle as a wrapper and initialize the KeyStore as follows: KeyStore.getInstance("BKS" , "SC").

Be aware that not all KeyStores properly protect the keys stored in the KeyStore files.

Reference

- owasp-mastg Data Storage Methods Overview Older KeyStore Implementations

Rulebook

- *On older Android OS, store keys by BouncyCastle KeyStore (Recommended)*

2.1.5 Key Chain

The KeyChain class is used to store and retrieve system-wide private keys and their corresponding certificates (chain). The user will be prompted to set a lock screen pin or password to protect the credential storage if something is being imported into the KeyChain for the first time. Note that the KeyChain is system-wide, every application can access the materials stored in the KeyChain.

Inspect the source code to determine whether native Android mechanisms identify sensitive information. Sensitive information should be encrypted, not stored in clear text. For sensitive information that must be stored on the device, several API calls are available to protect the data via the KeyChain class. Complete the following steps:

- Make sure that the app is using the Android KeyStore and Cipher mechanisms to securely store encrypted information on the device. Look for the patterns AndroidKeystore, import java.security.KeyStore, import javax.crypto.Cipher, import java.security.SecureRandom, and corresponding usages.

- Use the store(OutputStream stream, char[] password) function to store the KeyStore to disk with a password. Make sure that the password is provided by the user, not hard-coded.

Reference

- owasp-mastg Data Storage Methods Overview Keychain

Rulebook

- *Prompt users to set a lock screen pin or password to protect certificate storage when importing into Keychain for the first time (Required)*
- *Determine if native Android mechanisms identify sensitive information (Required)*

2.1.6 Storing a Cryptographic Key: Techniques

To mitigate unauthorized use of keys on the Android device, Android KeyStore lets apps specify authorized uses of their keys when generating or importing the keys. Once made, authorizations cannot be changed.

Storing a Key - from most secure to least secure:

- the key is stored in hardware-backed Android KeyStore
- all keys are stored on server and are available after strong authentication
- master key is stored on server and use to encrypt other keys, which are stored in Android SharedPreferences
- the key is derived each time from a strong user provided passphrase with sufficient length and salt
- the key is stored in software implementation of Android KeyStore
- master key is stored in software implementation of Android Keystore and used to encrypt other keys, which are stored in SharedPreferences
- [not recommended] all keys are stored in SharedPreferences
- [not recommended] hardcoded encryption keys in the source code
- [not recommended] predictable obfuscation function or key derivation function based on stable attributes
- [not recommended] stored generated keys in public places (like /sdcard/)

Reference

- owasp-mastg Data Storage Methods Overview Storing a Cryptographic Key: Techniques

Rulebook

- *Encryption key storage method (Required)*

2.1.6.1 Storing Keys Using Hardware-backed Android KeyStore

You can use the hardware-backed Android KeyStore if the device is running Android 7.0 (API level 24) and above with available hardware component (Trusted Execution Environment (TEE) or a Secure Element (SE)). You can even verify that the keys are hardware-backed by using the guidelines provided for the secure implementation of Key Attestation. If a hardware component is not available and/or support for Android 6.0 (API level 23) and below is required, then you might want to store your keys on a remote server and make them available after authentication.

Reference

- owasp-mastg Data Storage Methods Overview Storing Keys Using Hardware-backed Android KeyStore

2.1.6.2 Storing Keys on the Server

It is possible to securely store keys on a key management server, however the app needs to be online to decrypt the data. This might be a limitation for certain mobile app use cases and should be carefully thought through as this becomes part of the architecture of the app and might highly impact usability.

Reference

- owasp-mastg Data Storage Methods Overview Storing Keys on the Server

2.1.6.3 Deriving Keys from User Input

Deriving a key from a user provided passphrase is a common solution (depending on which Android API level you use), but it also impacts usability, might affect the attack surface and could introduce additional weaknesses.

Each time the application needs to perform a cryptographic operation, the user's passphrase is needed. Either the user is prompted for it every time, which isn't an ideal user experience, or the passphrase is kept in memory as long as the user is authenticated. Keeping the passphrase in memory is not a best-practice as any cryptographic material must only be kept in memory while it is being used. Zeroing out a key is often a very challenging task as explained in "*Cleaning out Key Material*".

Additionally, consider that keys derived from a passphrase have their own weaknesses. For instance, the passwords or passphrases might be reused by the user or easy to guess. Please refer to the *Testing Cryptography chapter* for more information.

Reference

- owasp-mastg Data Storage Methods Overview Deriving Keys from User Input

2.1.6.4 Cleaning out Key Material

The key material should be cleared out from memory as soon as it is not need anymore. There are certain limitations of realibly cleaning up secret data in languages with garbage collector (Java) and immutable strings (Kotlin). Java Cryptography Architecture Reference Guide suggests using char[] instead of String for storing sensitive data, and nullify array after usage.

Note that some ciphers do not properly clean up their byte-arrays. For instance, the AES Cipher in BouncyCastle does not always clean up its latest working key leaving some copies of the byte-array in memory. Next, BigInteger based keys (e.g. private keys) cannot be removed from the heap nor zeroed out without additional effort. Clearing byte array can be achieved by writing a wrapper which implements Destroyable.

Reference

- owasp-mastg Data Storage Methods Overview Cleaning out Key Material

Rulebook

- *Key material must be erased from memory as soon as it is no longer needed (Required)*

2.1.6.5 Storing Keys using Android KeyStore API

More user-friendly and recommended way is to use the Android KeyStore API system (itself or through KeyChain) to store key material. If it is possible, hardware-backed storage should be used. Otherwise, it should fallback to software implementation of Android Keystore. However, be aware that the AndroidKeyStore API has been changed significantly throughout various versions of Android. In earlier versions, the AndroidKeyStore API only supported storing public/private key pairs (e.g., RSA). Symmetric key support has only been added since Android 6.0 (API level 23). As a result, a developer needs to handle the different Android API levels to securely store symmetric keys.

Reference

- owasp-mastg Data Storage Methods Overview Storing Keys using Android KeyStore API

Rulebook

- *Save key material (Recommended)*

2.1.6.6 Storing keys by encrypting them with other keys

In order to securely store symmetric keys on devices running on Android 5.1 (API level 22) or lower, we need to generate a public/private key pairs. We encrypt the symmetric key using the public key and store the private key in the AndroidKeyStore. The encrypted symmetric key can encoded using base64 and stored in the SharedPreferences. Whenever we need the symmetric key, the application retrieves the private key from the AndroidKeyStore and decrypts the symmetric key.

Envelope encryption, or key wrapping, is a similar approach that uses symmetric encryption to encapsulate key material. Data encryption keys (DEKs) can be encrypted with key encryption keys (KEKs) which are securely stored. Encrypted DEKs can be stored in SharedPreferences or written to files. When required, the application reads the KEK, then decrypts the DEK. Refer to OWASP Cryptographic Storage Cheat Sheet to learn more about encrypting cryptographic keys.

Also, as the illustration of this approach, refer to the EncryptedSharedPreferences from androidx.security.crypto package.

Reference

- owasp-mastg Data Storage Methods Overview Storing keys by encrypting them with other keys

Rulebook

- *Generate public/private key pairs for secure storage of symmetric keys under Android OS 5.1 (Required)*
- *EncryptedSharedPreferences usage (Recommended)*

2.1.6.7 Insecure options to store keys

A less secure way of storing encryption keys, is in the SharedPreferences of Android. When SharedPreferences are used, the file is only readable by the application that created it. However, on rooted devices any other application with root access can simply read the SharedPreference file of other apps. This is not the case for the AndroidKeyStore. Since AndroidKeyStore access is managed on kernel level, which needs considerably more work and skill to bypass without the AndroidKeyStore clearing or destroying the keys.

The last three options are to use hardcoded encryption keys in the source code, having a predictable obfuscation function or key derivation function based on stable attributes, and storing generated keys in public places like /sdcard/. Hardcoded encryption keys are an issue since this means every instance of the application uses the same encryption key. An attacker can reverse-engineer a local copy of the application in order to extract the cryptographic key, and use that key to decrypt any data which was encrypted by the application on any device.

Next, when you have a predictable key derivation function based on identifiers which are accessible to other applications, the attacker only needs to find the KDF and apply it to the device in order to find the key. Lastly, storing encryption keys publicly also is highly discouraged as other applications can have permission to read the public partition and steal the keys.

Reference

- owasp-mastg Data Storage Methods Overview Insecure options to store keys

Rulebook

- *Do not use insecure encryption key storage methods (Required)*

2.1.7 Third Party libraries

There are several different open-source libraries that offer encryption capabilities specific for the Android platform.

- Java AES Crypto - A simple Android class for encrypting and decrypting strings.
- SQL Cipher - SQLCipher is an open source extension to SQLite that provides transparent 256-bit AES encryption of database files.
- Secure Preferences - Android Shared preference wrapper than encrypts the keys and values of Shared Preferences.
- Themis - A cross-platform high-level cryptographic library that provides same API across many platforms for securing data during authentication, storage, messaging, etc.

Please keep in mind that as long as the key is not stored in the KeyStore, it is always possible to easily retrieve the key on a rooted device and then decrypt the values you are trying to protect.

Reference

- owasp-mastg Data Storage Methods Overview Third Party libraries

Rulebook

- *Encryption with third-party libraries (Deprecated)*

2.1.8 Rulebook

1. *Verify that keys are stored inside security hardware (Recommended)*
2. *How to use StrongBox (Recommended)*
3. *For secure key authentication, provide a certificate in the device by means of a challenge received from the server (Recommended)*
4. *Check for secure implementation of key authentication in terms of security analysis (Required)*
5. *On older Android OS, store keys by BouncyCastle KeyStore (Recommended)*
6. *Prompt users to set a lock screen pin or password to protect certificate storage when importing into Keychain for the first time (Required)*
7. *Determine if native Android mechanisms identify sensitive information (Required)*
8. *Encryption key storage method (Required)*
9. *Key material must be erased from memory as soon as it is no longer needed (Required)*
10. *Save key material (Recommended)*
11. *Generate public/private key pairs for secure storage of symmetric keys under Android OS 5.1 (Required)*
12. *EncryptedSharedPreferences usage (Recommended)*
13. *Do not use insecure encryption key storage methods (Required)*
14. *Encryption with third-party libraries (Deprecated)*

2.1.8.1 Verify that keys are stored inside security hardware (Recommended)

It is possible to check if the key is stored inside the security hardware (by checking if KeyInfo.isinsideSecureHardware returns true).

The method to check is as follows. Since isinsideSecureHardware is deprecated for API level 31 and above Apps targeting API level 31 or higher should use getSecurityLevel.

```
      KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
      keyStore.load(null);
      SecretKey secretKey = (SecretKey) keyStore.getKey("ALIAS", null);
      SecretKeyFactory secretKeyFactory = SecretKeyFactory.
↪getInstance(KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");
      KeyInfo keyInfo = (KeyInfo) secretKeyFactory.getKeySpec(secretKey, KeyInfo.
↪class);
      if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.S) {
         int securityLevel = keyInfo.getSecurityLevel();
      } else {
         boolean isInsideSecureHardware = keyInfo.isInsideSecureHardware();
      }
```

If this is not noted, the following may occur.

- The key could be read/written by the user and misused.

2.1.8.2 How to use StrongBox (Recommended)

Devices running Android 9 (API level 28) or later can include the StrongBox Keymaster module. This is an implementation of the Keymaster HAL that resides in a hardware security module with its own CPU, Secure Storage, true random number generator, and a mechanism to combat package tampering. To use this feature, the setIsStrongBoxBacked Builder class or KeyProtection. To ensure that the StrongBox is used at runtime, make sure that isInsideSecureHardware returns true and the system does not throw a StrongBoxUnavailableException. Note that isinsideSecureHardware is abolished in API level 31 and getSecurityLevel is recommended.

```
      KeyGenParameterSpec builder = new KeyGenParameterSpec.Builder("ALIAS",␣
↪KeyProperties.PURPOSE_VERIFY)
              .setIsStrongBoxBacked(true)
              .build();
```

If this is not noted, the following may occur.

- The key could be read/written by the user and misused.

2.1.8.3 For secure key authentication, provide a certificate in the device by means of a challenge received from the server (Recommended)

Although key authentication can be implemented only on the client side, it is recommended that it be implemented on the server side for more secure authentication. In this case, the client needs to call the setAttestationChallenge API with the challenge received from the server, use the KeyStore.getCertificateChain method to obtain a certificate chain, and provide the certificate to the server. The server performs key authentication using the provided certificate.

The following is a sample code of the above process.

```
final KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);
keyStore.deleteEntry(KEYSTORE_ALIAS_SAMPLE);
final KeyGenParameterSpec.Builder builder = new KeyGenParameterSpec.
↪Builder(KEYSTORE_ALIAS_SAMPLE,
    KeyProperties.PURPOSE_SIGN | KeyProperties.PURPOSE_VERIFY)
    .setAlgorithmParameterSpec(new ECGenParameterSpec(AttestationProtocol.EC_
↪CURVE))
    .setDigests(AttestationProtocol.KEY_DIGEST)
```

(continues on next page)

(continued from previous page)

```
    .setAttestationChallenge(challenge);
AttestationProtocol.generateKeyPair(KEY_ALGORITHM_EC, builder.build());
final Certificate[] certs = keyStore.getCertificateChain(KEYSTORE_ALIAS_SAMPLE);
```

If this is not noted, the following may occur.

- The device receiving the challenge may not be able to prove that it is the device in question and may not be able to perform secure key authentication.

2.1.8.4 Check for secure implementation of key authentication in terms of security analysis (Required)

From a security analysis perspective, analysts should perform the following checks for secure implementation of key authentication

- Ensure that key authentication is not fully implemented on the client side. If fully implemented, the same can be easily circumvented by tampering with the application or hooking a method.
- Ensure that the server uses a random challenge when initiating key authentication. Failure to do so results in an insecure implementation and makes it vulnerable to replay attacks. Also, checks should be made regarding the randomness of the challenge.
- Check that the server verifies the integrity of the key authentication response; failure to do so is an insecure implementation and should be addressed.
- Verify that the server performs basic checks on certificates in the chain, including integrity, trustworthiness, and validity; failure to do so constitutes an insecure implementation and should be addressed.

If this is not noted, the following may occur.

- Secure key authentication cannot be guaranteed.

2.1.8.5 On older Android OS, store keys by BouncyCastle KeyStore (Recommended)

Older versions of Android do not include KeyStore, but do include the KeyStore interface of JCA (Java Cryptography Architecture). By using a KeyStore that implements this interface, the confidentiality and integrity of keys stored in the KeyStore can be ensured. BouncyCastle KeyStore (BKS) is recommended.

An implementation of KeyStore using BouncyCastle KeyStore (BKS) is described below. "BKS" is the KeyStore name (BouncyCastle Keystore) and "BC" means provider (BouncyCastle). Using SpongyCastle as a wrapper, it is also possible to initialize a KeyStore as follows.

Note that not all KeyStores adequately protect the keys stored in the KeyStore file.

```
KeyStore.getInstance("BKS", "SC")
```

If this is not noted, the following may occur.

- The confidentiality and integrity of the keys used may not be ensured.

2.1.8.6 Prompt users to set a lock screen pin or password to protect certificate storage when importing into KeyChain for the first time (Required)

When importing something into KeyChain for the first time, users are prompted to set a lock screen pin or password to protect certificate storage. It should be noted that KeyChain is system-wide and all applications have access to materials stored in KeyChain.

If this is violated, the following may occur.

- Information imported into the Keychain is available at the system level and could be used for unintended purposes if the device is used by a third party.

2.1.8.7 Determine if native Android mechanisms identify sensitive information (Required)

Examine the source code to determine if native Android mechanisms identify sensitive information. Sensitive information should be encrypted and should not be stored in plain text. For sensitive information that must be stored on the device, several API calls are available to protect the data via the KeyChain class. Complete the following steps.

- Verify that the app is storing encrypted information on the device.
- Verify that the app is using the Android KeyStore and Cipher mechanisms to securely store encrypted information on the device.

  Look for the following patterns.

  - AndroidKeystore
  - import java.security.KeyStore
  - import javax.crypto.Cipher
  - import java.security.SecureRandom, and corresponding usages
- store(OutputStream stream, char[] password) function to store the KeyStore to disk with a password. Make sure that the password is not hard-coded, but provided by the user. Sample code is shown below.

```
public static void main(String args[]) throws Exception {
    char[] oldpass = args[0].toCharArray();
    char[] newpass = args[1].toCharArray();
    String name = "mykeystore";
    FileInputStream in = new FileInputStream(name);
    KeyStore ks = KeyStore.getInstance(/*jca name*/);
    ks.load(in, oldpass);
    in.close();
    FileOutputStream output = new FileOutputStream(name);
    ks.store(output, newpass);
    output.close();
}
```

The following sample code shows how to install credentials in the KeyChain class.

```
private val launcher = registerForActivityResult(
    contract = ActivityResultContracts.StartActivityForResult()
) { result ->
    //...
}

fun startPiyoActivity() {
    val bis = BufferedInputStream(assets.open(/*PKCS12 filename*/))
    val keychain = ByteArray(bis.available())
    bis.read(keychain)
    val installIntent = Keychain.createInstallIntent()
    installIntent.putExtra(Keychain.EXTRA_PKCS12, keychain)
    installIntent.putExtra(Keychain.EXTRA_NAME, /*alias*/)
    val intent = Intent(this, /*Activity name*/::class.java)
    launcher.launch(intent)
}
```

If this is violated, the following may occur.

- Sensitive information may be stored in plain text and leaked to third parties.

2.1.8.8 Encryption key storage method (Required)

The following are secure and insecure methods of storing encryption keys.

Recommended.

The following are the recommended methods of key storage in order of security.

- Store the key in the Android KeyStore stored in hardware.

  The following is a sample code to save a key to Android KeyStore.

```
      KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
      keyStore.load(null);
      SecretKey secretKey = (SecretKey) keyStore.getKey("ALIAS", null);
      SecretKeyFactory secretKeyFactory = SecretKeyFactory.
↪getInstance(KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");
      KeyInfo keyInfo = (KeyInfo) secretKeyFactory.getKeySpec(secretKey,␣
↪KeyInfo.class);
      if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.
↪S) {
         int securityLevel = keyInfo.getSecurityLevel();
      } else {
         boolean isInsideSecureHardware = keyInfo.isInsideSecureHardware();
      }
```

- All keys are stored on the server and made available after strong authentication. * See "*Storing Keys on the Server*" for details and precautions.
- The master key is stored on the server and used to encrypt other keys stored in Android' s SharedPreferences. * See "*Storing Keys on the Server*" for details and precautions.
- Ensure that the key has sufficient length and Salt is derived each time from a strong passphrase provided by the user. * See "*Deriving Keys from User Input*" for details and precautions.
- Store the key in the software implementation of Android KeyStore. * See "*Cleaning out Key Material*" and "*Storing Keys using Android KeyStore API*" for details and precautions.
- The master key is stored in the software implementation of the Android Keystore and used to encrypt other keys stored in SharedPreferences. * See "*Cleaning out Key Material*" and "*Storing Keys using Android KeyStore API*" for details and precautions.

Deprecated

- Save all keys in SharedPreferences.
- Hardcode keys into source code.
- Predictable obfuscation or key derivation functions based on stable attributes.
- Store generated keys in a public location (e.g. /sdcard/).

If this is violated, the following may occur.

- Unauthorized use of keys on Android devices.

2.1.8.9 Key material must be erased from memory as soon as it is no longer needed (Required)

Key material should be erased from memory as soon as it is no longer needed. Languages that use garbage collectors (Java) or immutable strings (Kotlin) have certain limitations in actually cleaning up confidential data. The Java Cryptography Architecture Reference Guide suggests using char[] instead of String to store confidential data and nulling the array after use.

Example in Java:

```
        // Salt
        byte[] salt = new SecureRandom().nextBytes(/*salt*/);

        // Iteration count
        int count = 1000;

        // Create PBE parameter set
        pbeParamSpec = new PBEParameterSpec(salt, count);

        // Prompt user for encryption password.
        // Collect user password as char array, and convert
        // it into a SecretKey object, using a PBE key
        // factory.
        char[] password = /*cleartext string*/.toCharArray();
        pbeKeySpec = new PBEKeySpec(password);
        keyFac = SecretKeyFactory.getInstance(/*algorithm*/);
        SecretKey pbeKey = keyFac.generateSecret(pbeKeySpec);

        // Create PBE Cipher
        Cipher pbeCipher = Cipher.getInstance(/*algorithm*/);

        // Initialize PBE Cipher with key and parameters
        pbeCipher.init(Cipher.ENCRYPT_MODE, pbeKey, pbeParamSpec);

        // Our cleartext
        byte[] cleartext = /*cleartext string*/.getBytes();

        // Encrypt the cleartext
        byte[] ciphertext = pbeCipher.doFinal(cleartext);


        cleartext = null;
        ciphertext = null;
```

Note that some ciphers do not properly clean up byte arrays. For example, BouncyCastle' s AES cipher does not always clean up the latest working key, but leaves some copies of the byte array in memory. Second, BigInteger-based keys (e.g., secret keys) cannot be deleted or zeroed from the heap without additional effort. Clearing the byte array can be accomplished by creating a wrapper that implements Destroyable.

Example in Java:

```
KeyStore.PasswordProtection ks = new KeyStore.PasswordProtection("password".
↪toCharArray());
ks.destroy();

if(ks.isDestroyed()){
   cleartext = null;
   ciphertext = null;
}
```

If this is violated, the following may occur.

- Key material in memory may be used for other purposes.
- Languages that use garbage collectors (Java) or immutable strings (Kotlin) may not be cleaned up.

2.1.8.10 Save key material (Recommended)

A more user-friendly and recommended method is to use the Android KeyStore API system (by itself or via KeyChain) system (by itself or via KeyChain) to store key material.

The storage method using the KeyStore API is as follows.

Example in Java:

```
  // save my secret key
   javax.crypto.SecretKey mySecretKey;
   KeyStore.SecretKeyEntry skEntry =
       new KeyStore.SecretKeyEntry(mySecretKey);
   ks.setEntry("secretKeyAlias", skEntry, protParam);
```

If this is violated, the following may occur.

- Keys cannot be stored securely and may be misused.

2.1.8.11 Generate public/private key pairs for secure storage of symmetric keys under Android OS 5.1 (Required)

To securely store symmetric keys on devices with Android 5.1 (API level 22) or lower, a public/private key pair must be generated. The public key is used to encrypt the symmetric key, and the private key is stored in the Android KeyStore. The encrypted symmetric key can be encoded in base64 and stored in SharedPreferences. Whenever a symmetric key is needed, the application retrieves the private key from the Android KeyStore and decrypts the symmetric key.

* No sample code due to conceptual rule.

If this is violated, the following may occur.

- Symmetric keys cannot be stored securely and may be misused.

2.1.8.12 EncryptedSharedPreferences usage (Recommended)

The following is how the EncryptedSharedPreferences in the androidx.security.crypto package should be handled when encrypting a key with another key. The corresponding method is as follows.

Example in Java:

```
MasterKey masterKey = new MasterKey.Builder(context)
    .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
    .build();

SharedPreferences sharedPreferences = EncryptedSharedPreferences.create(
    context,
    "secret_shared_prefs",
    masterKey,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
);

// use the shared preferences and editor as you normally would
SharedPreferences.Editor editor = sharedPreferences.edit();
```

If this is violated, the following may occur.

- Symmetric keys cannot be stored securely and may be misused.

2.1.8.13 Do not use insecure encryption key storage methods (Required)

The use of SharedPreferences or hardcodes, which are considered insecure methods of storing encryption keys, is dangerous and should not be used.

Insecure methods of storing encryption keys are as follows:

- Storing in SharedPreferences; on a rooted device, other applications with root access can easily read other applications' SharedPreferences files.
- Hard-code to source code. An attacker can reverse engineer a local copy of the application, extract the encryption key, and use that key to decrypt data encrypted by the application on any device
- If there is a predictable key derivation function based on identifiers accessible by other applications, an attacker can find the key simply by finding the KDF and applying it to the device.
- Store the encryption key in public. Not recommended since other applications have the privilege to read the public partition and can steal the key.

2.1.8.14 Encryption with third-party libraries (Deprecated)

The following open source libraries exist that provide encryption capabilities specific to the Android platform. While the libraries are useful, their use is discouraged because it is always possible for a rooted device to easily retrieve the key and decrypt the value it is trying to protect, unless the key is stored in a KeyStore.

- Java AES Crypto: A simple Android class for encrypting and decrypting strings.
- SQL Cipher: SQLCipher is an open source extension to SQLite that provides transparent 256-bit AES encryption of database files.
- Secure Preferences: Android Shared preference wrapper provides encryption of Shared Preferences keys and values.
- Themis: A cross-platform high-level encryption library that provides the same API on many platforms to protect authentication, storage, messaging, and other data.

The reasons for this deprecation are as follows.

- Unless the key is stored in the KeyStore, it is always possible to easily retrieve the key on a rooted device and decrypt the value you are trying to protect.

2.2 MSTG-STORAGE-2

No sensitive data should be stored outside of the app container or system credential storage facilities.

2.2.1 Internal Storage

You can save files to the device' s internal storage. Files saved to internal storage are containerized by default and cannot be accessed by other apps on the device. When the user uninstalls your app, these files are removed. The following code snippets would persistently store sensitive data to internal storage.

Example for Java:

```
FileOutputStream fos = null;
try {
   fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
   fos.write(test.getBytes());
   fos.close();
} catch (FileNotFoundException e) {
   e.printStackTrace();
} catch (IOException e) {
```

(continues on next page)

(continued from previous page)

```
    e.printStackTrace();
}
```

Example for Kotlin:

```
var fos: FileOutputStream? = null
fos = openFileOutput("FILENAME", Context.MODE_PRIVATE)
fos.write(test.toByteArray(Charsets.UTF_8))
fos.close()
```

You should check the file mode to make sure that only the app can access the file. You can set this access with MODE_PRIVATE. Modes such as MODE_WORLD_READABLE (deprecated) and MODE_WORLD_WRITEABLE (deprecated) may pose a security risk.

Search for the class FileInputStream to find out which files are opened and read within the app.

Reference

- owasp-mastg Data Storage Methods Overview Internal Storage

Rulebook

- *Store sensitive data in the app container or in the system's credentials storage function (Required)*

2.2.2 External Storage

Every Android-compatible device supports shared external storage. This storage may be removable (such as an SD card) or internal (non-removable). Files saved to external storage are world-readable. The user can modify them when USB mass storage is enabled. You can use the following code snippets to persistently store sensitive information to external storage as the contents of the file password.txt.

Example for Java:

```
File file = new File (Environment.getExternalFilesDir(), "password.txt");
String password = "SecretPassword";
FileOutputStream fos;
    fos = new FileOutputStream(file);
    fos.write(password.getBytes());
    fos.close();
```

Example for Kotlin:

```
val password = "SecretPassword"
val path = context.getExternalFilesDir(null)
val file = File(path, "password.txt")
file.appendText(password)
```

The file will be created and the data will be stored in a clear text file in external storage once the activity has been called.

It's also worth knowing that files stored outside the application folder (data/data/<package-name>/) will not be deleted when the user uninstalls the application. Finally, it's worth noting that the external storage can be used by an attacker to allow for arbitrary control of the application in some cases. For more information: see the blog from Checkpoint.

Reference

- owasp-mastg Data Storage Methods Overview External Storage

2.2.3 SharedPreferences

The SharedPreferences API is commonly used to permanently save small collections of key-value pairs. Data stored in a SharedPreferences object is written to a plain-text XML file. The SharedPreferences object can be declared world-readable (accessible to all apps) or private. Misuse of the SharedPreferences API can often lead to exposure of sensitive data. Consider the following example:

Example for Java:

```
SharedPreferences sharedPref = getSharedPreferences("key", MODE_WORLD_READABLE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString("username", "administrator");
editor.putString("password", "supersecret");
editor.commit();
```

Example for Kotlin:

```
var sharedPref = getSharedPreferences("key", Context.MODE_WORLD_READABLE)
var editor = sharedPref.edit()
editor.putString("username", "administrator")
editor.putString("password", "supersecret")
editor.commit()
```

Once the activity has been called, the file key.xml will be created with the provided data. This code violates several best practices.

- The username and password are stored in clear text in /data/data/<package-name>/shared_prefs/key.xml.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <string name="username">administrator</string>
  <string name="password">supersecret</string>
</map>
```

- MODE_WORLD_READABLE allows all applications to access and read the contents of key.xml.

```
root@hermes:/data/data/sg.vp.owasp_mobile.myfirstapp/shared_prefs # ls -la
-rw-rw-r-- u0_a118    170 2016-04-23 16:51 key.xml
```

* Please note that MODE_WORLD_READABLE and MODE_WORLD_WRITEABLE were deprecated starting on API level 17. Although newer devices may not be affected by this, applications compiled with an android:targetSdkVersion value less than 17 may be affected if they run on an OS version that was released before Android 4.2 (API level 17).

Reference

- owasp-mastg Data Storage Methods Overview Shared Preferences

2.2.4 Databases

The Android platform provides a number of database options as aforementioned in the previous list. Each database option has its own quirks and methods that need to be understood.

Reference

- owasp-mastg Data Storage Methods Overview Databases

2.2.4.1 SQLite

SQLite Database (Unencrypted) SQLite is an SQL database engine that stores data in .db files. The Android SDK has built-in support for SQLite databases. The main package used to manage the databases is android.database.sqlite. For example, you may use the following code to store sensitive information within an activity:

Example in Java:

```
SQLiteDatabase notSoSecure = openOrCreateDatabase("privateNotSoSecure", MODE_
↪PRIVATE, null);
notSoSecure.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR,␣
↪Password VARCHAR);");
notSoSecure.execSQL("INSERT INTO Accounts VALUES('admin','AdminPass');");
notSoSecure.close();
```

Example in Kotlin:

```
var notSoSecure = openOrCreateDatabase("privateNotSoSecure", Context.MODE_PRIVATE,␣
↪null)
notSoSecure.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR,␣
↪Password VARCHAR);")
notSoSecure.execSQL("INSERT INTO Accounts VALUES('admin','AdminPass');")
notSoSecure.close()
```

Once the activity has been called, the database file privateNotSoSecure will be created with the provided data and stored in the clear text file /data/data/<package-name>/databases/privateNotSoSecure.

The database' s directory may contain several files besides the SQLite database:

- Journal files: These are temporary files used to implement atomic commit and rollback.
- Lock files: The lock files are part of the locking and journaling feature, which was designed to improve SQLite concurrency and reduce the writer starvation problem.

Sensitive information should not be stored in unencrypted SQLite databases.

SQLite Databases (Encrypted) With the library SQLCipher, SQLite databases can be password-encrypted.

Example in Java:

```
SQLiteDatabase secureDB = SQLiteDatabase.openOrCreateDatabase(database,
↪"password123", null);
secureDB.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR,Password␣
↪VARCHAR);");
secureDB.execSQL("INSERT INTO Accounts VALUES('admin','AdminPassEnc');");
secureDB.close();
```

Example in Kotlin:

```
var secureDB = SQLiteDatabase.openOrCreateDatabase(database, "password123", null)
secureDB.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR,Password␣
↪VARCHAR);")
secureDB.execSQL("INSERT INTO Accounts VALUES('admin','AdminPassEnc');")
secureDB.close()
```

Secure ways to retrieve the database key include:

- Asking the user to decrypt the database with a PIN or password once the app is opened (weak passwords and PINs are vulnerable to brute force attacks)
- Storing the key on the server and allowing it to be accessed from a web service only (so that the app can be used only when the device is online)

Reference

- owasp-mastg Data Storage Methods Overview SQLite Databases (Encrypted)

- owasp-mastg Data Storage Methods Overview SQLite Database (Unencrypted)

2.2.4.2 Firebase

Firebase is a development platform with more than 15 products, and one of them is Firebase Real-time Database. It can be leveraged by application developers to store and sync data with a NoSQL cloud-hosted database. The data is stored as JSON and is synchronized in real-time to every connected client and also remains available even when the application goes offline.

A misconfigured Firebase instance can be identified by making the following network call:

```
https://_firebaseProjectName_.firebaseio.com/.json
```

The firebaseProjectName can be retrieved from the mobile application by reverse engineering the application. Alternatively, the analysts can use Firebase Scanner, a python script that automates the task above as shown below:

```
python FirebaseScanner.py -p <pathOfAPKFile>

python FirebaseScanner.py -f <commaSeperatedFirebaseProjectNames>
```

Reference

- owasp-mastg Data Storage Methods Overview Firebase Real-time Databases

2.2.4.3 Realm

The Realm Database for Java is becoming more and more popular among developers. The database and its contents can be encrypted with a key stored in the configuration file.

```
//the getKey() method either gets the key from the server or from a KeyStore, or␣
↪is derived from a password.
RealmConfiguration config = new RealmConfiguration.Builder()
  .encryptionKey(getKey())
  .build();

Realm realm = Realm.getInstance(config);
```

If the database is not encrypted, you should be able to obtain the data. If the database is encrypted, determine whether the key is hard-coded in the source or resources and whether it is stored unprotected in shared preferences or some other location.

Reference

- owasp-mastg Data Storage Methods Overview Realm Databases

2.2.5 Rulebook

1. *Store sensitive data in the app container or in the system' s credentials storage function (Required)*

2.2.5.1 Store sensitive data in the app container or in the system' s credentials storage function (Required)

In order to securely store sensitive data, it is necessary to encrypt data using keys managed by the system' s credentials storage function (Android Keystore) and to ensure that encrypted data is stored in the application container (internal storage). For information on how to use the Android Keystore, see "*Encryption key storage method (Required)*" .

In order to avoid external reads, it must be implemented so that only the application can read and write files in internal storage. One method of creating files in internal storage is to use streams. In this method, "Context#openFileOutput" is called to obtain a "FileOutputStream" object to access a file in the filesDir directory. If the specified file does not exist, a new file is created.

In a call to Context#openFileOutput, the file mode must be specified. The file mode specified determines the read/write range of the created file.

The following are the main file modes.

- MODE_PRIVATE
- MODE_WORLD_READABLE
- MODE_WORLD_WRITEABLE

The following is an example of a Context#openFileOutput call. Note that on devices with Android 7.0 (API level 24) or later, if MODE_PRIVATE is not specified for the file mode, SecurityException will occur when calling.

```
String filename = "myfile";
String fileContents = "Hello world!";
try (FileOutputStream fos = context.openFileOutput(filename, Context.MODE_
↪PRIVATE)) {
    fos.write(fileContents.toByteArray());
}
```

File mode setting MODE_PRIVATE In default mode, the created file can be accessed by the calling application or by all applications sharing the same user ID.

```
public static final int MODE_PRIVATE
```

File mode setting MODE_WORLD_READABLE All other applications will have read access to the created file. Note that the use of MODE_WORLD_READABLE has been deprecated since API level 17.

```
public static final int MODE_WORLD_READABLE
```

File mode setting MODE_WORLD_WRITEABLE All other applications will have read access to the created file. Note that the use of MODE_WORLD_WRITEABLE has been deprecated since API level 17.

```
public static final int MODE_WORLD_WRITEABLE
```

If this is violated, the following may occur.

- Sensitive data can be read by other applications or third parties.

2.3 MSTG-STORAGE-3

No sensitive data is written to application logs.

2.3.1 Log Output

This test case focuses on identifying any sensitive application data within both system and application logs. The following checks should be performed:

- Analyze source code for logging related code.
- Check application data directory for log files.
- Gather system messages and logs and analyze for any sensitive data.

As a general recommendation to avoid potential sensitive application data leakage, logging statements should be removed from production releases unless deemed necessary to the application or explicitly identified as safe, e.g. as a result of a security audit.

Reference

- owasp-mastg Testing Logs for Sensitive Data (MSTG-STORAGE-3) Overview

2.3.1.1 File Write

Applications will often use the Log Class and Logger Class to create logs. To discover this, you should audit the application's source code for any such logging classes. These can often be found by searching for the following keywords:

- Functions and classes, such as:
  - android.util.Log
  - Log.d | Log.e | Log.i | Log.v | Log.w | Log.wtf
  - Logger
- Keywords and system output:
  - System.out.print | System.err.print
  - logfile
  - logging
  - logs

Reference

- owasp-mastg Testing Logs for Sensitive Data (MSTG-STORAGE-3) Static Analysis

Rulebook

- *When outputting logs, do not include confidential information in the output content (Required)*

2.3.1.2 Logcat Output

Use all the mobile app functions at least once, then identify the application's data directory and look for log files (/data/data/<package-name>). Check the application logs to determine whether log data has been generated; some mobile applications create and store their own logs in the data directory.

Many application developers still use System.out.println or printStackTrace instead of a proper logging class. Therefore, your testing strategy must include all output generated while the application is starting, running and closing. To determine what data is directly printed by System.out.println or printStackTrace, you can use Logcat as explained in the chapter "Basic Security Testing", section "Monitoring System Logs".

Remember that you can target a specific app by filtering the Logcat output as follows:

```
adb logcat | grep "$(adb shell ps | grep <package-name> | awk '{print $2}')"
```

* If you already know the app PID you may give it directly using – pid flag.

You may also want to apply further filters or regular expressions (using logcat's regex flags -e <expr>, – regex=<expr> for example) if you expect certain strings or patterns to come up in the logs.

Reference

- owasp-mastg Testing Logs for Sensitive Data (MSTG-STORAGE-3) Dynamic Analysis

2.3.1.3 Deletion of logging functions by ProGuard

While preparing the production release, you can use tools like ProGuard (included in Android Studio). To determine whether all logging functions from the android.util.Log class have been removed, check the ProGuard configuration file (proguard-rules.pro) for the following options (according to this example of removing logging code and this article about enabling ProGuard in an Android Studio project):

```
-assumenosideeffects class android.util.Log
{
  public static boolean isLoggable(java.lang.String, int);
  public static int v(...);
```

(continues on next page)

(continued from previous page)

```
  public static int i(...);
  public static int w(...);
  public static int d(...);
  public static int e(...);
  public static int wtf(...);
}
```

Note that the example above only ensures that calls to the Log class' methods will be removed. If the string that will be logged is dynamically constructed, the code that constructs the string may remain in the bytecode. For example, the following code issues an implicit StringBuilder to construct the log statement:

Example in Java:

```
Log.v("Private key tag", "Private key [byte format]: " + key);
```

Example in Kotlin:

```
Log.v("Private key tag", "Private key [byte format]: $key")
```

The compiled bytecode, however, is equivalent to the bytecode of the following log statement, which constructs the string explicitly:

Example in Java:

```
Log.v("Private key tag", new StringBuilder("Private key [byte format]: ").
↪append(key.toString()).toString());
```

Example in Kotlin:

```
Log.v("Private key tag", StringBuilder("Private key [byte format]: ").append(key).
↪toString())
```

ProGuard guarantees removal of the Log.v method call. Whether the rest of the code (new StringBuilder ⋯) will be removed depends on the complexity of the code and the ProGuard version.

This is a security risk because the (unused) string leaks plain text data into memory, which can be accessed via a debugger or memory dumping.

Unfortunately, no silver bullet exists for this issue, but one option would be to implement a custom logging facility that takes simple arguments and constructs the log statements internally.c

```
SecureLog.v("Private key [byte format]: ", key);
```

Then configure ProGuard to strip its calls.

Reference

- owasp-mastg Testing Logs for Sensitive Data (MSTG-STORAGE-3) Static Analysis

2.3.2 Rulebook

1. *When outputting logs, do not include confidential information in the output content (Required)*

2.3.2.1 When outputting logs, do not include confidential information in the output content (Required)

When outputting logs, it must be ensured that the output does not contain sensitive information.

Common log output classes include the following.

- Log
- Logger

Log Class

Log.v(), Log.d(), Log.i(), Log.w(), and Log.e() methods are included in the android.util package. The written logs can be viewed on Logcat.

Each method is classified by log level. The following is a list of log levels and their associated methods.

Table 2.3.2.1.1 List of Log Levels and Log Methods

No	Log Level	Methods
1	DEBUG	Log.d
2	ERROR	Log.e
3	INFO	Log.i
4	VERBOSE	Log.v
5	WARN	Log.w
6	What a Terrible Failure	Log.wtf

The following is an example of log output code by the Log class.

```
private static final String TAG = "MyActivity";
Log.v(TAG, "index=" + i);
```

Logger Class

A class included in java.util.logging for logging output, used to log messages for a specific system or application component. It is typically named using a hierarchical, dot-delimited namespace. The Logger name can be any string, but should usually be based on the package or class name of the component being logged (e.g., java.net or javax.swing).

Below is an example of the log output code by the Logger class.

```
class DiagnosisMessages {
  static String systemHealthStatus() {
    // collect system health information
    ...
  }
}
...
logger.log(Level.FINER, DiagnosisMessages.systemHealthStatus());
```

If this is violated, the following may occur.

- Third parties will be able to read confidential information.

2.4 MSTG-STORAGE-4

No sensitive data is shared with third parties unless it is a necessary part of the architecture.

2.4.1 Application Data Sharing

Sensitive information might be leaked to third parties by several means, which include but are not limited to the following:

Reference

- owasp-mastg Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4) Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4) Overview

2.4.1.1 Third-party Services Embedded in the App

The features these services provide can involve tracking services to monitor the user's behavior while using the app, selling banner advertisements, or improving the user experience.

The downside is that developers don't usually know the details of the code executed via third-party libraries. Consequently, no more information than is necessary should be sent to a service, and no sensitive information should be disclosed.

Most third-party services are implemented in two ways:

- with a standalone library
- with a full SDK

Static Analtsis To determine whether API calls and functions provided by the third-party library are used according to best practices, review their source code, requested permissions and check for any known vulnerabilities (see "*Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5)*".

All data that's sent to third-party services should be anonymized to prevent exposure of PII (Personal Identifiable Information) that would allow the third party to identify the user account. No other data (such as IDs that can be mapped to a user account or session) should be sent to a third party.

Dynamic Analysis Check all requests to external services for embedded sensitive information. To intercept traffic between the client and server, you can perform dynamic analysis by launching a man-in-the-middle (MITM) attack with Burp Suite Professional or OWASP ZAP. Once you route the traffic through the interception proxy, you can try to sniff the traffic that passes between the app and server. All app requests that aren't sent directly to the server on which the main function is hosted should be checked for sensitive information, such as PII in a tracker or ad service.

Reference

- owasp-mastg Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4) Third-party Services Embedded in the App
- owasp-mastg Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4) Third-party Services Embedded in the App Static Analysis
- owasp-mastg Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4) Third-party Services Embedded in the App Dynamic Analysis

Rulebook

- *Do not share unnecessarily confidential information to third-party libraries (Required)*

2.4.1.2 App Notifications

It is important to understand that notifications should never be considered private. When a notification is handled by the Android system it is broadcasted system-wide and any application running with a NotificationListenerService can listen for these notifications to receive them in full and may handle them however it wants.

There are many known malware samples such as Joker, and Alien which abuses the NotificationListenerService to listen for notifications on the device and then send them to attacker-controlled C2 infrastructure. Commonly this is done in order to listen for two-factor authentication (2FA) codes that appear as notifications on the device which are then sent to the attacker. A safer alternative for the user would be to use a 2FA application that does not generate notifications.

Furthermore there are a number of apps on the Google Play Store that provide notification logging, which basically logs locally any notifications on the Android system. This highlights that notifications are in no way private on Android and accessible by any other app on the device.

For this reason all notification usage should be inspected for confidential or high risk information that could be used by malicious applications.

Static Analtsis Search for any usage of the NotificationManager class which might be an indication of some form of notification management. If the class is being used, the next step would be to understand how the application is generating the notifications and which data ends up being shown.

Dynamic Analysis Run the application and start tracing all calls to functions related to the notifications creation, e.g. setContentTitle or setContentText from NotificationCompat.Builder. Observe the trace in the end and evaluate if it contains any sensitive information which another app might have eavesdropped.

Reference

- owasp-mastg Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4) App Notifications
- owasp-mastg Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4) App Notifications Static Analysis
- owasp-mastg Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4) App Notifications Dynamic Analysis

Rulebook

- *Do not include confidential information in the notification (Required)*

2.4.2 Rulebook

1. *Do not share unnecessarily confidential information to third-party libraries (Required)*
2. *Do not include confidential information in the notification (Required)*

2.4.2.1 Do not share unnecessarily confidential information to third-party libraries (Required)

When using a third-party library, make sure that no non-essential confidential information is set as a parameter to be passed to the library. If non-essential confidential information is set, be aware that it may be used maliciously in the library' s internal processing.

Since the library may be required for communication, if the above concerns are considered, it is necessary to encrypt confidential information using an encryption method decided between the server and client in advance and pass it to the library, etc.

If this is violated, the following may occur.

- May be exploited in the processing of third-party libraries.

2.4.2.2 Do not include confidential information in the notification (Required)

The NotificationManager class is used to notify users of events that have occurred.

The components of the notification (display content) are specified in the NotificationCompat.Builder object. Builder class provides methods for specifying the components of a notification. The following is an example of a method for specification.

- setContentTitle: Specifies the title (first line) of the notification in a standard notification.
- setContentText: In a standard notification, specifies the text of the notification (second line).

When using notifications, note that setContentTitle and setContentText are not set with sensitive information.

The following is an example of source code that specifies the components of a notification to the NotificationCompat.Builder class and displays the notification using the NotificationManager class.

```
var builder = NotificationCompat.Builder(this, CHANNEL_ID)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle(textTitle)
        .setContentText(textContent)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)
with(NotificationManagerCompat.from(this)) {
    // Pass notificationID and builder.build()
    notify(notificationID, builder.build())
}
```

If this is violated, the following may occur.

- Third parties will be able to read confidential information.

2.5 MSTG-STORAGE-5

The keyboard cache is disabled on text inputs that process sensitive data.

2.5.1 Automatic entry of confidential data

When users type in input fields, the software automatically suggests data. This feature can be very useful for messaging apps. However, the keyboard cache may disclose sensitive information when the user selects an input field that takes this type of information.

Static Analysis In the layout definition of an activity, you can define TextViews that have XML attributes. If the XML attribute android:inputType is given the value textNoSuggestions, the keyboard cache will not be shown when the input field is selected. The user will have to type everything manually.

```
<EditText
     android:id="@+id/KeyBoardCache"
     android:inputType="textNoSuggestions" />
```

The code for all input fields that take sensitive information should include this XML attribute to disable the keyboard suggestions.

Dynamic Analysis Start the app and click in the input fields that take sensitive data. If strings are suggested, the keyboard cache has not been disabled for these fields.

Reference

- owasp-mastg Determining Whether the Keyboard Cache Is Disabled for Text Input Fields MSTG-STORAGE-5)

Rulebook

- *Implement code for all input fields for sensitive information to disable keyboard suggestions (Required)*

- *All input field layouts for sensitive information should be implemented to disable keyboard suggestions (Required)*

2.5.2 Rulebook

1. *Implement code for all input fields for sensitive information to disable keyboard suggestions (Required)*
2. *All input field layouts for sensitive information should be implemented to disable keyboard suggestions (Required)*

2.5.2.1 Implement code for all input fields for sensitive information to disable keyboard suggestions (Required)

Use the EditText class for text input and modification. When defining a text editing widget, the android.R.styleable#TextView_inputType attribute must be set.

In the code for fields where confidential information is to be entered, set the TYPE_TEXT_FLAG_NO_SUGGESTIONS flag to the inputType attribute. However, if the field is for a password or pin, set the TYPE_TEXT_VARIATION_PASSWORD flag to the inputType attribute for masking (see *Input Text*).

Below is an example of a code that sets TYPE_TEXT_FLAG_NO_SUGGESTIONS as a flag for the inputType attribute on the code.

```
val editText1: EditText = findViewById(R.id.editText1)
editText1.apply {
    inputType = InputType.TYPE_TEXT_FLAG_NO_SUGGESTIONS
}
```

It is also necessary to make sure that the cache is not overwritten with a value that would re-enable it.

If this is violated, the following may occur.

- Third parties will be able to read confidential information.

2.5.2.2 All input field layouts for sensitive information should be implemented to disable keyboard suggestions (Required)

In the layout of a field for inputting confidential information (EditText), set "textNoSuggestions" to the inputType attribute. However, if the field is for a password or pin, set the "textPassword" to the inputType attribute for masking (see *Input Text*).

Below is an example of code that sets "textNoSuggestions" as the inputType attribute on the code.

```
<EditText
      android:id="@+id/KeyBoardCache"
      android:inputType="textNoSuggestions" />
```

If this is violated, the following may occur.

- Third parties will be able to read confidential information.

2.6 MSTG-STORAGE-6

No sensitive data is exposed via IPC mechanisms.

2.6.1 Access to sensitive data via ContentProvider

As part of Android' s IPC mechanisms, content providers allow an app' s stored data to be accessed and modified by other apps. If not properly configured, these mechanisms may leak sensitive data.

Static Analysis The first step is to look at AndroidManifest.xml to detect content providers exposed by the app. You can identify content providers by the <provider> element. Complete the following steps:

- Determine whether the value of the export tag (android:exported) is "true" . Even if it is not, the tag will be set to "true" automatically if an <intent-filter> has been defined for the tag. If the content is meant to be accessed only by the app itself, set android:exported to "false" . If not, set the flag to "true" and define proper read/write permissions.
- Determine whether the data is being protected by a permission tag (android:permission). Permission tags limit exposure to other apps.
- Determine whether the android:protectionLevel attribute has the value signature. This setting indicates that the data is intended to be accessed only by apps from the same enterprise (i.e., signed with the same key). To make the data accessible to other apps, apply a security policy with the <permission> element and set a proper android:protectionLevel. If you use android:permission, other applications must declare corresponding <uses-permission> elements in their manifests to interact with your content provider. You can use the android:grantUriPermissions attribute to grant more specific access to other apps; you can limit access with the <grant-uri-permission> element.

Inspect the source code to understand how the content provider is meant to be used. Search for the following keywords:

- android.content.ContentProvider
- android.database.Cursor
- android.database.sqlite
- .query
- .update
- .delete

* To avoid SQL injection attacks within the app, use parameterized query methods, such as query, update, and delete. Be sure to properly sanitize all method arguments; for example, the selection argument could lead to SQL injection if it is made up of concatenated user input.

If you expose a content provider, determine whether parameterized query methods (query, update, and delete) are being used to prevent SQL injection. If so, make sure all their arguments are properly sanitized.

We will use the vulnerable password manager app Sieve as an example of a vulnerable content provider.

Inspect the Android Manifest Identify all defined <provider> elements:

```
<provider
      android:authorities="com.mwr.example.sieve.DBContentProvider"
      android:exported="true"
      android:multiprocess="true"
      android:name=".DBContentProvider">
    <path-permission
          android:path="/Keys"
          android:readPermission="com.mwr.example.sieve.READ_KEYS"
          android:writePermission="com.mwr.example.sieve.WRITE_KEYS"
     />
</provider>
<provider
      android:authorities="com.mwr.example.sieve.FileBackupProvider"
      android:exported="true"
      android:multiprocess="true"
      android:name=".FileBackupProvider"
/>
```

As shown in the AndroidManifest.xml above, the application exports two content providers. Note that one path ("/Keys") is protected by read and write permissions.

Inspect the source code Inspect the query function in the DBContentProvider.java file to determine whether any sensitive information is being leaked:

Example in Java:

```
public Cursor query(final Uri uri, final String[] array, final String s, final␣
↪String[] array2, final String s2) {
    final int match = this.sUriMatcher.match(uri);
    final SQLiteQueryBuilder sqLiteQueryBuilder = new SQLiteQueryBuilder();
    if (match >= 100 && match < 200) {
        sqLiteQueryBuilder.setTables("Passwords");
    }
    else if (match >= 200) {
        sqLiteQueryBuilder.setTables("Key");
    }
    return sqLiteQueryBuilder.query(this.pwdb.getReadableDatabase(), array, s,␣
↪array2, (String)null, (String)null, s2);
}
```

Example in Kotlin:

```
fun query(uri: Uri?, array: Array<String?>?, s: String?, array2: Array<String?>?,␣
↪s2: String?): Cursor {
        val match: Int = this.sUriMatcher.match(uri)
        val sqLiteQueryBuilder = SQLiteQueryBuilder()
        if (match >= 100 && match < 200) {
            sqLiteQueryBuilder.tables = "Passwords"
        } else if (match >= 200) {
            sqLiteQueryBuilder.tables = "Key"
        }
        return sqLiteQueryBuilder.query(this.pwdb.getReadableDatabase(), array, s,␣
↪array2, null as String?, null as String?, s2)
    }
```

Here we see that there are actually two paths, "/Keys" and "/Passwords" , and the latter is not being protected in the manifest and is therefore vulnerable.

When accessing a URI, the query statement returns all passwords and the path Passwords/. We will address this in the "Dynamic Analysis" section and show the exact URI that is required.

Dynamic Analysis Testing Content Providers To dynamically analyze an application' s content providers, first enumerate the attack surface: pass the app' s package name to the Drozer module app.provider.info:

```
dz> run app.provider.info -a com.mwr.example.sieve
  Package: com.mwr.example.sieve
  Authority: com.mwr.example.sieve.DBContentProvider
  Read Permission: null
  Write Permission: null
  Content Provider: com.mwr.example.sieve.DBContentProvider
  Multiprocess Allowed: True
  Grant Uri Permissions: False
  Path Permissions:
  Path: /Keys
  Type: PATTERN_LITERAL
  Read Permission: com.mwr.example.sieve.READ_KEYS
  Write Permission: com.mwr.example.sieve.WRITE_KEYS
  Authority: com.mwr.example.sieve.FileBackupProvider
  Read Permission: null
  Write Permission: null
  Content Provider: com.mwr.example.sieve.FileBackupProvider
```

(continues on next page)

(continued from previous page)

```
  Multiprocess Allowed: True
  Grant Uri Permissions: False
```

In this example, two content providers are exported. Both can be accessed without permission, except for the /Keys path in the DBContentProvider. With this information, you can reconstruct part of the content URIs to access the DBContentProvider (the URIs begin with content://).

To identify content provider URIs within the application, use Drozer' s scanner.provider.finduris module. This module guesses paths and determines accessible content URIs in several ways:

```
dz> run scanner.provider.finduris -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Unable to Query content://com.mwr.example.sieve.DBContentProvider/
...
Unable to Query content://com.mwr.example.sieve.DBContentProvider/Keys
Accessible content URIs:
content://com.mwr.example.sieve.DBContentProvider/Keys/
content://com.mwr.example.sieve.DBContentProvider/Passwords
content://com.mwr.example.sieve.DBContentProvider/Passwords/
```

Once you have a list of accessible content providers, try to extract data from each provider with the app.provider.query module:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/
↪Passwords/ --vertical
_id: 1
service: Email
username: incognitoguy50
password: PSFjqXIMVa5NJFudgDuuLVgJYFD+8w== (Base64 - encoded)
email: incognitoguy50@gmail.com
```

You can also use Drozer to insert, update, and delete records from a vulnerable content provider:

- Insert record

```
dz> run app.provider.insert content://com.vulnerable.im/messages
              --string date 1331763850325
              --string type 0
              --integer _id 7
```

- Update record

```
dz> run app.provider.update content://settings/secure
              --selection "name=?"
              --selection-args assisted_gps_enabled
              --integer value 0
```

- Delete record

```
dz> run app.provider.delete content://settings/secure
              --selection "name=?"
              --selection-args my_setting
```

SQL Injection in Content Providers The Android platform promotes SQLite databases for storing user data. Because these databases are based on SQL, they may be vulnerable to SQL injection. You can use the Drozer module app.provider.query to test for SQL injection by manipulating the projection and selection fields that are passed to the content provider:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/
↪Passwords/ --projection "'"
unrecognized token: "' FROM Passwords" (code 1): , while compiling: SELECT ' FROM␣
```

(continues on next page)

(continued from previous page)

```
↪Passwords

dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/
↪Passwords/ --selection "'"
unrecognized token: "')" (code 1): , while compiling: SELECT * FROM Passwords␣
↪WHERE (')
```

If an application is vulnerable to SQL Injection, it will return a verbose error message. SQL Injection on Android may be used to modify or query data from the vulnerable content provider. In the following example, the Drozer module app.provider.query is used to list all the database tables:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/
↪Passwords/ --projection "*
FROM SQLITE_MASTER WHERE type='table';--"
| type  | name              | tbl_name          | rootpage | sql               |
| table | android_metadata | android_metadata | 3        | CREATE TABLE ... |
| table | Passwords         | Passwords         | 4        | CREATE TABLE ... |
| table | Key               | Key               | 5        | CREATE TABLE ... |
```

SQL Injection may also be used to retrieve data from otherwise protected tables:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/
↪Passwords/ --projection "* FROM Key;--"
| Password | pin |
| thisismypassword | 9876 |
```

You can automate these steps with the scanner.provider.injection module, which automatically finds vulnerable content providers within an app:

```
dz> run scanner.provider.injection -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Injection in Projection:
  content://com.mwr.example.sieve.DBContentProvider/Keys/
  content://com.mwr.example.sieve.DBContentProvider/Passwords
  content://com.mwr.example.sieve.DBContentProvider/Passwords/
Injection in Selection:
  content://com.mwr.example.sieve.DBContentProvider/Keys/
  content://com.mwr.example.sieve.DBContentProvider/Passwords
  content://com.mwr.example.sieve.DBContentProvider/Passwords/
```

File System Based Content Providers Content providers can provide access to the underlying filesystem. This allows apps to share files (the Android sandbox normally prevents this).You can use the Drozer modules app.provider.read and app.provider.download to read and download files, respectively, from exported file-based content providers.These content providers are susceptible to directory traversal, which allows otherwise protected files in the target application' s sandbox to be read.

```
dz> run app.provider.download content://com.vulnerable.app.FileProvider/../../../..
↪/../../../../data/data/com.vulnerable.app/database.db /home/user/database.db
Written 24488 bytes
```

Use the scanner.provider.traversal module to automate the process of finding content providers that are susceptible to directory traversal:

```
dz> run scanner.provider.traversal -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Vulnerable Providers:
  content://com.mwr.example.sieve.FileBackupProvider/
  content://com.mwr.example.sieve.FileBackupProvider
```

Note that adb can also be used to query content providers:

```
$ adb shell content query --uri content://com.owaspomtg.vulnapp.provider.
↪CredentialProvider/credentials
Row: 0 id=1, username=admin, password=StrongPwd
Row: 1 id=2, username=test, password=test
...
```

Reference

- owasp-mastg Determining Whether Sensitive Stored Data Has Been Exposed via IPC Mechanisms (MSTG-STORAGE-6)

Rulebook

- *Set appropriate access permissions for ContentProvider (Required)*
- *Take measures against SQL injection when using SQL databases (Required)*
- *Take measures against directory traversal when using ContentProvider (Required)*

2.6.2 Rulebook

1. *Set appropriate access permissions for ContentProvider (Required)*
2. *Take measures against SQL injection when using SQL databases (Required)*
3. *Take measures against directory traversal when using ContentProvider (Required)*

2.6.2.1 Set appropriate access permissions for ContentProvider (Required)

All ContentProviders in the app must be defined in the <provider> element in AndroidManifest.xml. If undefined, the system will not recognize the ContentProvider and will not execute it.

Declare only the ContentProvider that is part of the target application, and do not declare any ContentProvider that is used within the target application but is part of another application.

The following is an example of a definition of the <provider> element in AndroidManifest.xml

```
<provider android:authorities="list"
          android:directBootAware=["true" | "false"]
          android:enabled=["true" | "false"]
          android:exported=["true" | "false"]
          android:grantUriPermissions=["true" | "false"]
          android:icon="drawable resource"
          android:initOrder="integer"
          android:label="string resource"
          android:multiprocess=["true" | "false"]
          android:name="string"
          android:permission="string"
          android:process="string"
          android:readPermission="string"
          android:syncable=["true" | "false"]
          android:writePermission="string" >
    . . .
</provider>
```

Content providers must be configured appropriately considering the data to be accessed.

Tags that configure whether ContentProvider can be used by other apps android:exported This tag allows you to configure whether other apps can use ContentProvider. The following are the possible settings

- true：Other apps can use ContentProvider. Any application can access ContentProvider using the content URI of ContentProvider according to the permissions specified for ContentProvider.

- false：Other apps cannot use ContentProvider. When android:exported=" false" is set, access to ContentProvider is limited to the target application.When this is set, apps that can access the ContentProvider are limited to apps that have the same user ID (UID) as the ContentProvider, or apps that have been granted temporary access rights by the android:grantUriPermissions tag.

Since this tag was introduced in API level 17, all devices with API level 16 or lower will behave as if this tag were set to "true" . If android:targetSdkVersion is set to 17 or higher, the default value is "false" for devices with API level 17 or higher.

* Note that even if exported is false, if an <intent-filter> is defined, the situation is the same as if exported were set to true.

Tag android:permission to set the <permission> element name needed for ContentProvider data read/write.

Sets the <permission> element name required by the client when reading/writing ContentProvider data.This attribute is useful for setting a single authorization for both reading and writing. However, the android:readPermission, android:writePermission, and android:grantUriPermissions attributes take precedence over this attribute. If the android:readPermission attribute is also set, access to query ContentProvider is controlled. If the android:writePermission attribute is set, access to change data in ContentProvider is controlled.

By setting the level of protection to the android:protectionLevel tag within the <permission> element, one can specify the risks that may be included in the authorization and the steps that the system must follow when deciding whether to grant the authorization to the requesting app.

Each protection level specifies a basic authority type and protectionLevel. The following is a list of basic authority types

Table 2.6.2.1.1 List of protectionLevel Basic Authority Types

Basic Authority Type	Description.
normal	Default Value. Low-risk permissions that provide access to isolated app-level functionality to the requesting app.
dangerous	High-risk permissions that allow the requesting app to access personal data or control the device, which could negatively impact the user.
signature	Authority granted by the system only if the same certificate as the app declaring the authority is used to sign the requesting app.
signatureOrSystem	Authority granted by the system only to apps installed in a dedicated folder in the Android system image or apps signed using the same certificate as the app that declared the authority. Note that this is deprecated in API level 23.

Temporarily grant access to ContentProvider data tag android:grantUriPermissions

Sets whether users who do not have permission to access ContentProvider data can be granted such privileges. If granted, the restrictions imposed by the android:readPermission, android:writePermission, android:permission, and android:exported attributes are temporarily lifted. Set to "true" if permission can be granted, otherwise set to "false" . If set to "true" , permissions can be granted for any data in ContentProvider. If set to "false" , authorization can only be granted for the data subset (if any) listed in the sub-element. The default value is "false" .

If this is violated, the following may occur.

- Sensitive data may be unintentionally leaked to other apps.

2.6.2.2 Take measures against SQL injection when using SQL databases (Required)

When using SQL databases in ContentProvider, it is necessary to take measures against SQL injection.

The measures to be taken are described below.

1. If you do not need to expose ContentProvider to other apps:
   - In the manifest, change the / tag of the target ContentProvider and set it to android:exported=" false" . This will prevent other apps from sending intents to the target ContentProvider. *The android:permission attribute can also be set to the permission of the android:protectionLevel=" signature" to prevent apps written by other developers from sending intents to the target ContentProvider.
2. If you need to expose ContentProvider to other apps:

   The sql to be passed to the query() method is pre-validated, and unnecessary characters are escaped. Also, using ? as a substitutable parameter in a select clause and a separate array of select arguments reduces risk by binding user input directly to the query rather than interpreting it as part of the SQL statement. A sample code is shown below.

```
public boolean validateOrderDetails(String email, String orderNumber) {
    boolean result = false;

    // Proprietary validation check
    if (!validationParam(email, orderNumber)) {
        // For violation parameters
        return result;
    }

    Cursor cursor = db.rawQuery(
      "select * from purchases where EMAIL = ? and ORDER_NUMBER = ?",
      new String[]{email, orderNumber});
    if (cursor != null) {
        if (cursor.moveToFirst()) {
            result = true;
        }
        cursor.close();
    }
    return result;
}
```

If this is violated, the following may occur.

- SQL injection vulnerabilities may be exploited.

2.6.2.3 Take measures against directory traversal when using ContentProvider (Required)

When using ContentProvider, directory traversal must be addressed.

The measures to be taken are described below.

1. If you do not need to expose ContentProvider to other apps:
   - In the manifest, change the / tag of the target ContentProvider and set it to android:exported=" false" . This will prevent other apps from sending intents to the target ContentProvider.
   - The android:permission attribute can also be set to the permission of the android:protectionLevel=" signature" to prevent apps written by other developers from sending intents to the target ContentProvider.
2. If you need to expose ContentProvider to other apps:

   When the input to openFile contains path traversal characters, it must be configured correctly so that the app will never return an unexpected file. To do so, check the canonical path of the file. A sample code is shown below.

```
public ParcelFileDescriptor openFile (Uri uri, String mode) throws␣
↪FileNotFoundException {
    File f = new File(DIR, uri.getLastPathSegment());
    if (!f.getCanonicalPath().startsWith(DIR)) {
        throw new IllegalArgumentException();
    }
    return ParcelFileDescriptor.open(f, ParcelFileDescriptor.MODE_READ_ONLY);
}
```

If this is violated, the following may occur.

- Directory traversal vulnerabilities may be exploited.

2.7 MSTG-STORAGE-7

No sensitive data, such as passwords or pins, is exposed through the user interface.

2.7.1 Checking for Sensitive Data Disclosure Through the User Interface

Entering sensitive information when, for example, registering an account or making payments, is an essential part of using many apps. This data may be financial information such as credit card data or user account passwords. The data may be exposed if the app doesn't properly mask it while it is being typed.

In order to prevent disclosure and mitigate risks such as shoulder surfing you should verify that no sensitive data is exposed via the user interface unless explicitly required (e.g. a password being entered). For the data required to be present it should be properly masked, typically by showing asterisks or dots instead of clear text.

Carefully review all UI components that either show such information or take it as input. Search for any traces of sensitive information and evaluate if it should be masked or completely removed.

Reference

- owasp-mastg Checking for Sensitive Data Disclosure Through the User Interface (MSTG-STORAGE-7)

2.7.1.1 Input text

Static Analysis To make sure an application is masking sensitive user input, check for the following attribute in the definition of EditText:

```
android:inputType="textPassword"
```

With this setting, dots (instead of the input characters) will be displayed in the text field, preventing the app from leaking passwords or pins to the user interface.

Dynamic Analysis If the information is masked by, for example, replacing input with asterisks or dots, the app isn't leaking data to the user interface.

Reference

- owasp-mastg Checking for Sensitive Data Disclosure Through the User Interface (MSTG-STORAGE-7) Text Fields

Rulebook

- *Masking in fields for sensitive passwords and pins (Required)*

2.7.1.2 App Notifications

Static Analysis When statically assessing an application, it is recommended to search for any usage of the NotificationManager class which might be an indication of some form of notification management. If the class is being used, the next step would be to understand how the application is generating the notifications.

These code locations can be fed into the Dynamic Analysis section below, providing an idea of where in the application notifications may be dynamically generated.

Dynamic Analysis

To identify the usage of notifications run through the entire application and all its available functions looking for ways to trigger any notifications. Consider that you may need to perform actions outside of the application in order to trigger certain notifications.

While running the application you may want to start tracing all calls to functions related to the notifications creation, e.g. setContentTitle or setContentText from NotificationCompat.Builder. Observe the trace in the end and evaluate if it contains any sensitive information.

Reference

- owasp-mastg Checking for Sensitive Data Disclosure Through the User Interface (MSTG-STORAGE-7) App Notifications

Rulebook

- *Implement with an understanding of how the application generates notifications and which data to display (Required)*

2.7.2 Rulebook

1. *Masking in fields for sensitive passwords and pins (Required)*
2. *Implement with an understanding of how the application generates notifications and which data to display (Required)*

2.7.2.1 Masking in fields for sensitive passwords and pins (Required)

Passwords and pins, which are confidential information, can be leaked by being displayed. Therefore, they should be masked or hidden in the field.

Below is a method for masking input fields.

For layouts:.

```
   <EditText
        android:id= @+id/Password
        android:inputType="textPassword" />
```

For code:.

```
val editText1: EditText = findViewById(R.id.editText1)
editText1.apply {
    inputType = InputType.TYPE_TEXT_VARIATION_PASSWORD
}
```

If this is violated, the following may occur.

- Third parties will be able to read confidential information.

2.7.2.2 Implement with an understanding of how the application generates notifications and which data to display (Required)

Use the NotificationManager class to notify users of events that have occurred.

The components of the notification (display content) are specified in the NotificationCompat.Builder object. The NotificationCompat.Builder class provides methods for specifying the components of the notification. The following is an example of a method for specifying.

- setContentTitle：In a standard notification, specify the title (first line) of the notification.
- setContentText：In a standard notification, specify the text of the notification (second line).

The following is an example of source code that specifies the components of a notification to the NotificationCompat.Builder class and displays the notification using the NotificationManager class.

```
var builder = NotificationCompat.Builder(this, CHANNEL_ID)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle(textTitle)
        .setContentText(textContent)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)
with(NotificationManagerCompat.from(this)) {
    // Pass notificationID and builder.build()
    notify(notificationID, builder.build())
}
```

If this is violated, the following may occur.

- Third parties will be able to read confidential information.

2.8 MSTG-STORAGE-12

The app educates the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app.

2.8.1 Testing User Education on Data Privacy on the App Marketplace

At this point, we're only interested in knowing which privacy-related information is being disclosed by the developers and trying to evaluate if it seems reasonable (similarly as you'd do when testing for permissions).

It's possible that the developers are not declaring certain information that is indeed being collected and or shared, but that's a topic for a different test extending this one here. As part of this test, you are not supposed to provide privacy violation assurance.

Reference

- owasp-mastg Testing User Education (MSTG-STORAGE-12) Testing User Education Testing User Education on Data Privacy on the App Marketplace

2.8.2 Static Analysis

You can follow these steps:

1. Search for the app in the corresponding app marketplace (e.g. Google Play, App Store).
2. Go to the section "Privacy Details" (App Store) or "Safety Section" (Google Play).
3. Verify if there's any information available at all.

The test passes if the developer has compiled with the app marketplace guidelines and included the required labels and explanations. Store and provide the information you got from the app marketplace as evidence, so that you can later use it to evaluate potential violations of privacy or data protection.

Reference

- owasp-mastg Testing User Education (MSTG-STORAGE-12) Testing User Education Static Analysis

2.8.3 Dynamic analysis

As an optional step, you can also provide some kind of evidence as part of this test. For instance, if you're testing an iOS app you can easily enable app activity recording and export a Privacy Report containing detailed app access to different resources such as photos, contacts, camera, microphone, network connections, etc.

Doing this has actually many advantages for testing other MASVS categories. It provides very useful information that you can use to test network communication in MASVS-NETWORK or when testing app permissions in MASVS-PLATFORM. While testing these other categories you might have taken similar measurements using other testing tools. You can also provide this as evidence for this test.

Ideally, the information available should be compared against what the app is actually meant to do. However, that's far from a trivial task that could take from several days to weeks to complete depending on your resources and support from automated tooling. It also heavily depends on the app functionality and context and should be ideally performed on a white box setup working very closely with the app developers.

Reference

- owasp-mastg Testing User Education (MSTG-STORAGE-12) Testing User Education Dynamic analysis

2.8.4 Testing User Education on Security Best Practices

Testing this might be especially challenging if you intend to automate it. We recommend using the app extensively and try to answer the following questions whenever applicable:

- Fingerprint usage: when fingerprints are used for authentication providing access to high-risk transactions/information, does the app inform the user about potential issues when having multiple fingerprints of other people registered to the device as well?
- Rooting/Jailbreaking: when root or jailbreak detection is implemented, does the app inform the user of the fact that certain high-risk actions will carry additional risk due to the jailbroken/rooted status of the device?
- Specific credentials: when a user gets a recovery code, a password or a pin from the application (or sets one), does the app instruct the user to never share this with anyone else and that only the app will request it?
- Application distribution: in case of a high-risk application and in order to prevent users from downloading compromised versions of the application, does the app manufacturer properly communicate the official way of distributing the app (e.g. from Google Play)?
- Prominent Disclosure: in any case, does the app display prominent disclosure of data access, collection, use, and sharing? e.g. does the app use the Best practices for prominent disclosure and consent to ask for the permission on Android?

Reference

- owasp-mastg Testing User Education (MSTG-STORAGE-12) Testing User Education Testing User Education on Security Best Practices

Rulebook

- *Use the app extensively to answer questions about security best practices (Recommended)*

2.8.5 Rulebook

1. *Use the app extensively to answer questions about security best practices (Recommended)*

2.8.5.1 Use the app extensively to answer questions about security best practices (Recommended)

It is recommended that the app be used extensively to answer the following questions regarding security best practices

- Fingerprint usage: when fingerprints are used for authentication providing access to high-risk transactions/information, does the app inform the user about potential issues when having multiple fingerprints of other people registered to the device as well?
- Rooting/Jailbreaking: when root or jailbreak detection is implemented, does the app inform the user of the fact that certain high-risk actions will carry additional risk due to the jailbroken/rooted status of the device?
- Specific credentials: when a user gets a recovery code, a password or a pin from the application (or sets one), does the app instruct the user to never share this with anyone else and that only the app will request it?
- Application distribution: in case of a high-risk application and in order to prevent users from downloading compromised versions of the application, does the app manufacturer properly communicate the official way of distributing the app (e.g. from Google Play)?
- Prominent Disclosure: in any case, does the app display prominent disclosure of data access, collection, use, and sharing? e.g. does the app use the Best practices for prominent disclosure and consent to ask for the permission on Android?

If this is not noted, the following may occur.

- Confidential information is used in an unintended process.
- Third parties will be able to read confidential information.

3 Cryptography Requirements

3.1 MSTG-CRYPTO-1

The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption.

3.1.1 Problematic Encryption Configuration

3.1.1.1 Insufficient Key Length

Even the most secure encryption algorithm becomes vulnerable to brute-force attacks when that algorithm uses an insufficient key size.

Ensure that the key length fulfills accepted industry standards. In Japan, check the List of Cipher Specifications on the "e-Government Recommended Ciphers List" .

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Insufficient Key Length

Rulebook

- *Set key lengths that meet industry standards (Required)*

3.1.1.2 Symmetric Encryption with Hard-Coded Cryptographic Keys

The security of symmetric encryption and keyed hashes (MACs) depends on the secrecy of the key. If the key is disclosed, the security gained by encryption is lost. To prevent this, never store secret keys in the same place as the encrypted data they helped create. A common mistake is encrypting locally stored data with a static, hardcoded encryption key and compiling that key into the app. This makes the key accessible to anyone who can use a disassembler.

Hardcoded encryption key means that a key is:

- part of application resources
- value which can be derived from known values
- hardcoded in code

First, ensure that no keys or passwords are stored within the source code. This means you should check native code, JavaScript/Dart code, Java/Kotlin code on Android. Note that hard-coded keys are problematic even if the source code is obfuscated since obfuscation is easily bypassed by dynamic instrumentation.

If the app is using two-way TLS (both server and client certificates are validated), make sure that:

- The password to the client certificate isn't stored locally or is locked in the device Keychain.
- The client certificate isn't shared among all installations.

If the app relies on an additional encrypted container stored in app data, check how the encryption key is used. If a key-wrapping scheme is used, ensure that the master secret is initialized for each user or the container is re-encrypted with new key. If you can use the master secret or previous password to decrypt the container, check how password changes are handled.

Secret keys must be stored in secure device storage whenever symmetric cryptography is used in mobile apps. For more information on the platform-specific APIs, see the "Data Storage on Android" chapters.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Symmetric Encryption with Hard-Coded Cryptographic Keys

Rulebook

- *Do not store keys or passwords in source code (Required)*
- *Do not store client certificate passwords locally. Lock passwords in your device's keychain if you want to save them (Required)*
- *Client certificates are not shared among all installations (Required)*
- *If container dependent, verify how encryption keys are used (Required)*
- *Store private keys in secure device storage whenever symmetric encryption is used in mobile apps (Required)*

3.1.1.3 Weak Key Generation Functions

Cryptographic algorithms (such as symmetric encryption or some MACs) expect a secret input of a given size. For example, AES uses a key of exactly 16 bytes. A native implementation might use the user-supplied password directly as an input key. Using a user-supplied password as an input key has the following problems:

- If the password is smaller than the key, the full key space isn't used. The remaining space is padded (spaces are sometimes used for padding).
- A user-supplied password will realistically consist mostly of displayable and pronounceable characters. Therefore, only some of the possible 256 ASCII characters are used and entropy is decreased by approximately a factor of four.

Ensure that passwords aren't directly passed into an encryption function. Instead, the user-supplied password should be passed into a KDF to create a cryptographic key. Choose an appropriate iteration count when using password derivation functions. For example, NIST recommends an iteration count of at least 10,000 for PBKDF2 and for critical keys where user-perceived performance is not critical at least 10,000,000. For critical keys, it is recommended to consider implementation of algorithms recognized by Password Hashing Competition (PHC) like Argon2.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Weak Key Generation Functions

Rulebook

- *If using an encryption algorithm (such as symmetric encryption or some MACs), use a secret input of the specific size assumed (Required)*
- *User-supplied passwords are passed to KDF to create encryption keys (Required)*
- *If using a password derivation function, select the appropriate number of iterations (Required)*

3.1.1.4 Weak Random Number Generators

It is fundamentally impossible to produce truly random numbers on any deterministic device. Pseudo-random number generators (RNG) compensate for this by producing a stream of pseudo-random numbers - a stream of numbers that appear as if they were randomly generated. The quality of the generated numbers varies with the type of algorithm used. Cryptographically secure RNGs generate random numbers that pass statistical randomness tests, and are resilient against prediction attacks (e.g. it is statistically infeasible to predict the next number produced).

Mobile SDKs offer standard implementations of RNG algorithms that produce numbers with sufficient artificial randomness. We'll introduce the available APIs in the Android specific sections.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Weak Random Number Generators

Rulebook

- *Identify a standard implementation of the RNG algorithm that generates numbers with sufficient artificial randomness (Required)*

3.1.1.5 Custom Implementations of Cryptography

Inventing proprietary cryptographic functions is time consuming, difficult, and likely to fail. Instead, we can use well-known algorithms that are widely regarded as secure. Mobile operating systems offer standard cryptographic APIs that implement those algorithms.

Carefully inspect all the cryptographic methods used within the source code, especially those that are directly applied to sensitive data. All cryptographic operations should use standard cryptographic APIs for Android (we'll write about those in more detail in the platform-specific chapters). Any cryptographic operations that don't invoke standard routines from known providers should be closely inspected. Pay close attention to standard algorithms that have been modified. Remember that encoding isn't the same as encryption! Always investigate further when you find bit manipulation operators like XOR (exclusive OR).

At all implementations of cryptography, you need to ensure that the following always takes place:

- Worker keys (like intermediary/derived keys in AES/DES/Rijndael) are properly removed from memory after consumption or in case of error.
- The inner state of a cipher should be removed from memory as soon as possible.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Custom Implementations of Cryptography

Rulebook

- *All cryptography-related implementations properly manage memory state (Required)*
- *Use industry-standard cryptographic APIs provided by the OS (Required)*

3.1.1.6 Incorrect AES Configuration

Advanced Encryption Standard (AES) is the widely accepted standard for symmetric encryption in mobile apps. It's an iterative block cipher that is based on a series of linked mathematical operations. AES performs a variable number of rounds on the input, each of which involve substitution and permutation of the bytes in the input block. Each round uses a 128-bit round key which is derived from the original AES key.

As of this writing, no efficient cryptanalytic attacks against AES have been discovered. However, implementation details and configurable parameters such as the block cipher mode leave some margin for error.

Weak Block Cipher Mode

Block-based encryption is performed upon discrete input blocks (for example, AES has 128-bit blocks). If the plaintext is larger than the block size, the plaintext is internally split up into blocks of the given input size and encryption is

performed on each block. A block cipher mode of operation (or block mode) determines if the result of encrypting the previous block impacts subsequent blocks.

ECB (Electronic Codebook) divides the input into fixed-size blocks that are encrypted separately using the same key. If multiple divided blocks contain the same plaintext, they will be encrypted into identical ciphertext blocks which makes patterns in data easier to identify. In some situations, an attacker might also be able to replay the encrypted data.

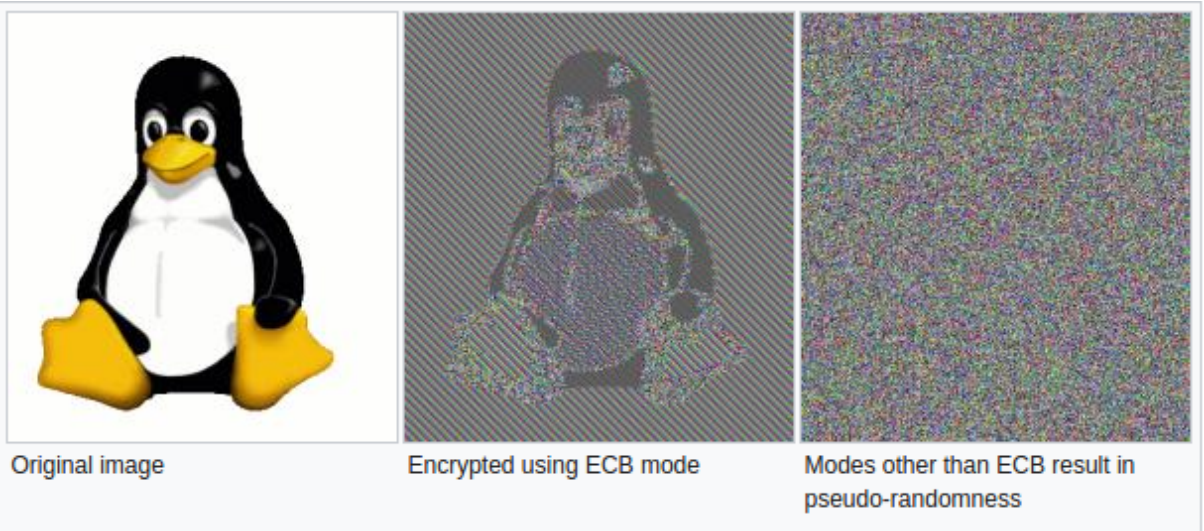


Fig 3.1.1.6.1 ECB Encryption Example

Verify that Cipher Block Chaining (CBC) mode is used instead of ECB. In CBC mode, plaintext blocks are XORed with the previous ciphertext block. This ensures that each encrypted block is unique and randomized even if blocks contain the same information. Please note that it is best to combine CBC with an HMAC and/or ensure that no errors are given such as "Padding error", "MAC error", "decryption failed" in order to be more resistant to a padding oracle attack.

When storing encrypted data, we recommend using a block mode that also protects the integrity of the stored data, such as Galois/Counter Mode (GCM). The latter has the additional benefit that the algorithm is mandatory for each TLSv1.2 implementation, and thus is available on all modern platforms.

For more information on effective block modes, see the NIST guidelines on block mode selection.

Predictable Initialization Vector

CBC, OFB, CFB, PCBC, GCM mode require an initialization vector (IV) as an initial input to the cipher. The IV doesn't have to be kept secret, but it shouldn't be predictable: it should be random and unique/non-repeatable for each encrypted message. Make sure that IVs are generated using a cryptographically secure random number generator. For more information on IVs, see Crypto Fail's initialization vectors article.

Pay attention to cryptographic libraries used in the code: many open source libraries provide examples in their documentations that might follow bad practices (e.g. using a hardcoded IV). A popular mistake is copy-pasting example code without changing the IV value.

Initialization Vectors in stateful operation modes

Please note that the usage of IVs is different when using CTR and GCM mode in which the initialization vector is often a counter (in CTR combined with a nonce). So here using a predictable IV with its own stateful model is exactly what is needed. In CTR you have a new nonce plus counter as an input to every new block operation. For example: for a 5120 bit long plaintext: you have 20 blocks, so you need 20 input vectors consisting of a nonce and counter. Whereas in GCM you have a single IV per cryptographic operation, which should not be repeated with the same key. See section 8 of the documentation from NIST on GCM for more details and recommendations of the IV.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Inadequate AES Configuration
- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Weak Block Cipher Mode
- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Predictable Initialization Vector
- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Initialization Vectors in stateful operation modes

Rulebook

- *To counter padding oracle attacks, CBC should not be combined with HMAC or generate errors such as padding errors, MAC errors, decryption failures, etc. (Required)*
- *When storing encrypted data, use a block mode such as Galois/Counter Mode (GCM) that also protects the integrity of the stored data (Recommended)*
- *IV is generated using a cryptographically secure random number generator (Required)*
- *Note that IV is used differently when using CTR and GCM modes, where the initialization vector is often a counter (Required)*

3.1.1.7 Padding Oracle Attacks due to Weaker Padding or Block Operation Implementations

In the old days, PKCS1.5 padding (in code: PKCS1Padding) was used as a padding mechanism when doing asymmetric encryption. This mechanism is vulnerable to the padding oracle attack. Therefore, it is best to use OAEP (Optimal Asymmetric Encryption Padding) captured in PKCS#1 v2.0 (in code: OAEPPadding, OAEPwithSHA-256andMGF1Padding, OAEPwithSHA-224andMGF1Padding, OAEPwithSHA-384andMGF1Padding, OAEPwithSHA-512andMGF1Padding). Note that, even when using OAEP, you can still run into an issue known best as the Mangers attack as described in the blog at Kudelskisecurity.

Note: AES-CBC with PKCS #5 has shown to be vulnerable to padding oracle attacks as well, given that the implementation gives warnings, such as “Padding error” , “MAC error” , or “decryption failed” . See The Padding Oracle Attack and The CBC Padding Oracle Problem for an example. Next, it is best to ensure that you add an HMAC after you encrypt the plaintext: after all a ciphertext with a failing MAC will not have to be decrypted and can be discarded.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Padding Oracle Attacks due to Weaker Padding or Block Operation Implementations

Rulebook

- *Use OAEP incorporated in PKCS#1 v2.0 as a padding mechanism for asymmetric encryption (Required)*

3.1.1.8 Protecting Keys in Storage and in Memory

When memory dumping is part of your threat model, then keys can be accessed the moment they are actively used. Memory dumping either requires root-access (e.g. a rooted device or jailbroken device) or it requires a patched application with Frida (so you can use tools like Fridump). Therefore it is best to consider the following, if keys are still needed at the device:

- Keys in a Remote Server: you can use remote Key vaults such as Amazon KMS or Azure Key Vault. For some use cases, developing an orchestration layer between the app and the remote resource might be a suitable option. For instance, a serverless function running on a Function as a Service (FaaS) system (e.g. AWS Lambda or Google Cloud Functions) which forwards requests to retrieve an API key or secret. There are other alternatives such as Amazon Cognito, Google Identity Platform or Azure Active Directory.

- Keys inside Secure Hardware-backed Storage: make sure that all cryptographic actions and the key itself remain in the Trusted Execution Environment (e.g. use Android Keystore). Refer to the Android Data Storage chapters for more information.
- Keys protected by Envelope Encryption: If keys are stored outside of the TEE / SE, consider using multi-layered encryption: an envelope encryption approach (see OWASP Cryptographic Storage Cheat Sheet, Google Cloud Key management guide, AWS Well-Architected Framework guide), or a HPKE approach to encrypt data encryption keys with key encryption keys.
- Keys in Memory: make sure that keys live in memory for the shortest time possible and consider zeroing out and nullifying keys after successful cryptographic operations, and in case of error. For general cryptocoding guidelines, refer to Clean memory of secret data. For more detailed information refer to sections Testing Memory for Sensitive Data respectively.

Note: given the ease of memory dumping, never share the same key among accounts and/or devices, other than public keys used for signature verification or encryption.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Protecting Keys in Storage and in Memory

RuleBook

- *Use key considering memory dump (Required)*
- *Do not share the same key across accounts or devices (Required)*

3.1.1.9 Key handling during transfer

When keys need to be transported from one device to another, or from the app to a backend, make sure that proper key protection is in place, by means of a transport keypair or another mechanism. Often, keys are shared with obfuscation methods which can be easily reversed. Instead, make sure asymmetric cryptography or wrapping keys are used. For example, a symmetric key can be encrypted with the public key from an asymmetric key pair.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Protecting Keys in Transport

Rulebook

- *Appropriate key protection by means of transport symmetric keys or other mechanisms (Required)*

3.1.2 Hardcoded symmetric encryption

This test case focuses on hardcoded symmetric cryptography as the only method of encryption. The following checks should be performed:

- identify all instances of symmetric cryptography
- for each identified instance verify if there are any hardcoded symmetric keys
- verify if hardcoded symmetric cryptography is not used as the only method of encryption

Reference

- owasp-mastg Testing Symmetric Cryptography (MSTG-CRYPTO-1) Overview

RuleBook

- *Do not use hardcoded symmetric encryption as the only method of encryption (Required)*

3.1.2.1 Static Analysis

Identify all the instances of symmetric key encryption in code and look for any mechanism which loads or provides a symmetric key. You can look for:

- symmetric algorithms (such as DES, AES, etc.)
- specifications for a key generator (such as KeyGenParameterSpec, KeyPairGeneratorSpec, KeyPairGenerator, KeyGenerator, KeyProperties, etc.)
- classes importing java.security.*, javax.crypto.*, android.security.*, android.security.keystore.*

For each identified instance verify if the used symmetric keys:

- are not part of the application resources
- cannot be derived from known values
- are not hardcoded in code

For each hardcoded symmetric key, verify that is not used in security-sensitive contexts as the only method of encryption.

As an example we illustrate how to locate the use of a hardcoded encryption key. First disassemble and decompile the app to obtain Java code, e.g. by using jadx.

Now search the files for the usage of the SecretKeySpec class, e.g. by simply recursively grepping on them or using jadx search function:

```
grep -r "SecretKeySpec"
```

This will return all classes using the SecretKeySpec class. Now examine those files and trace which variables are used to pass the key material. The figure below shows the result of performing this assessment on a production ready application. We can clearly locate the use of a static encryption key that is hardcoded and initialized in the static byte array Encrypt.keyBytes.

```
3  import javax.crypto.spec.*;
4  import javax.crypto.*;
5  import java.security.*;
6  import android.util.*;
7
8  public class Encrypt
9  {
10     private static byte[] keyBytes;
11
12     static {
13         Encrypt.keyBytes = new byte[] { 7, 3, 4, 5, 6, 7, 8, 9, 16, 17, 18, 9, 20, 21, 15, 1, 10, 11, 12, 13, 14,
14     }
15
16     public static String decrypt(final String s) throws Exception {
17         final SecretKeySpec secretKeySpec = new SecretKeySpec(Encrypt.keyBytes, "AES");
18         final Cipher instance = Cipher.getInstance("AES");
19         instance.init(2, secretKeySpec);
20         return new String(instance.doFinal(Base64.decode(s.getBytes(), 0)));
21     }
22
23     public static String encrypt(final String s) throws Exception {
24         final SecretKeySpec secretKeySpec = new SecretKeySpec(Encrypt.keyBytes, "AES");
25         final Cipher instance = Cipher.getInstance("AES");
26         instance.init(1, secretKeySpec);
27         return new String(Base64.encode(instance.doFinal(s.getBytes()), 0));
28     }
29 }
30
```

Fig 3.1.2.1.1 Identifying the use of hardcoded encryption keys

Reference

- owasp-mastg Testing Symmetric Cryptography (MSTG-CRYPTO-1) Static Analysis

3.1.2.2 Dynamic Analysis

You can use method tracing on cryptographic methods to determine input / output values such as the keys that are being used. Monitor file system access while cryptographic operations are being performed to assess where key material is written to or read from. For example, monitor the file system by using the API monitor of RMS - Runtime Mobile Security.

Reference

- owasp-mastg Testing Symmetric Cryptography (MSTG-CRYPTO-1) Dynamic Analysis

RuleBook

- *Perform a method trace of the encryption method to verify the write-to or read-from source of the key material (Required)*

3.1.3 Rulebook

1. *Set key lengths that meet industry standards (Required)*
2. *Do not store keys or passwords in source code (Required)*
3. *Do not store client certificate passwords locally. Lock passwords in your device' s keychain if you want to save them (Required)*
4. *Client certificates are not shared among all installations (Required)*
5. *If container dependent, verify how encryption keys are used (Required)*
6. *Store private keys in secure device storage whenever symmetric encryption is used in mobile apps (Required)*
7. *If using an encryption algorithm (such as symmetric encryption or some MACs), use a secret input of the specific size assumed (Required)*
8. *User-supplied passwords are passed to KDF to create encryption keys (Required)*
9. *If using a password derivation function, select the appropriate number of iterations (Required)*
10. *Identify a standard implementation of the RNG algorithm that generates numbers with sufficient artificial randomness (Required)*
11. *All cryptography-related implementations properly manage memory state (Required)*
12. *Use industry-standard cryptographic APIs provided by the OS (Required)*
13. *To counter padding oracle attacks, CBC should not be combined with HMAC or generate errors such as padding errors, MAC errors, decryption failures, etc. (Required)*
14. *When storing encrypted data, use a block mode such as Galois/Counter Mode (GCM) that also protects the integrity of the stored data (Recommended)*
15. *IV is generated using a cryptographically secure random number generator (Required)*
16. *Note that IV is used differently when using CTR and GCM modes, where the initialization vector is often a counter (Required)*
17. *Use OAEP incorporated in PKCS#1 v2.0 as a padding mechanism for asymmetric encryption (Required)*
18. *Use key considering memory dump (Required)*
19. *Do not share the same key across accounts or devices (Required)*
20. *Appropriate key protection by means of transport symmetric keys or other mechanisms (Required)*
21. *Do not use hardcoded symmetric encryption as the only method of encryption (Required)*
22. *Perform a method trace of the encryption method to verify the write-to or read-from source of the key material (Required)*

3.1.3.1 Set key lengths that meet industry standards (Required)

Ensure that the key length fulfills accepted industry standards. In Japan, check the List of Cipher Specifications on the "e-Government Recommended Ciphers List". Even the most secure encryption algorithms are vulnerable to brute force attacks if insufficient key sizes are used.

* No sample code due to conceptual rule.

If this is violated, the following may occur.

- Become vulnerable to brute force attacks.

3.1.3.2 Do not store keys or passwords in source code (Required)

Since obfuscation is easily bypassed by dynamic instrumentation, hardcoded keys are problematic even if the source code is obfuscated. Therefore, do not store keys or passwords within the source code (native code, JavaScript/Dart code, Java/Kotlin code).

* No sample code due to deprecated rules.

If this is violated, the following may occur.

- Keys and passwords are leaked.

3.1.3.3 Do not store client certificate passwords locally. Lock passwords in your device's keychain if you want to save them (Required)

If the app uses bi-directional TLS (both server and client certificates are verified), do not store the client certificate password locally. Or lock it in the device's Keychain.

See the rulebook below for sample code.

Rulebook

- *Prompt users to set a lock screen pin or password to protect certificate storage when importing into KeyChain for the first time (Required)*
- *Determine if native Android mechanisms identify sensitive information (Required)*

If this is violated, the following may occur.

- The password is read and misused by a third party.

3.1.3.4 Client certificates are not shared among all installations (Required)

If the app uses bi-directional TLS (both server and client certificates are verified), client certificates are not shared among all installations.

* No sample code due to deprecated rules.

If this is violated, the following may occur.

- The password is read and abused by other applications.

3.1.3.5 If container dependent, verify how encryption keys are used (Required)

If the app relies on an encrypted container stored within the app's data, identify how the encryption key is used.

When using the key wrap method

Confirm the following

- The master secret of each user must be initialized
- That the container is re-encrypted with the new key

If the container can be decrypted using the master secret or a previous password

Check how password changes are handled.

* No sample code due to conceptual rules.

If this is violated, the following may occur.

- A password or master secret is used for purposes other than those for which it was intended.

3.1.3.6 Store private keys in secure device storage whenever symmetric encryption is used in mobile apps (Required)

Whenever symmetric encryption is used in a mobile app, the private key must be stored in secure device storage. See "*Storing a Cryptographic Key: Techniques*" for information on how to store private keys on the Android platform.

Rulebook

- *Encryption key storage method (Required)*

If this is violated, the following may occur.

- The private key is read by another application or third party.

3.1.3.7 If using an encryption algorithm (such as symmetric encryption or some MACs), use a secret input of the specific size assumed (Required)

When using encryption algorithms (such as symmetric encryption and some MACs), it is necessary to use a secret input of the specific size expected. For example, AES uses a key of exactly 16 bytes.

Native implementations may use user-supplied passwords directly as input keys. When using user-supplied passwords as input keys, the following problems exist.

- If the password is smaller than the key, the full key space is not used. The remaining spaces are padded (sometimes spaces are used for padding).
- User-supplied passwords, in reality, consist mostly of characters that can be displayed and pronounced. Thus, only a fraction of the 256 ASCII characters are used, reducing entropy by about a factor of four.

* No sample code due to conceptual rules.

If this is violated, the following may occur.

- A vulnerable key is generated.

3.1.3.8 User-supplied passwords are passed to KDF to create encryption keys (Required)

If the encryption function is used,User-supplied passwords should be passed to KDF to create the encryption key. The password should not be passed directly to the encryption function. Instead, the user-supplied password should be passed to the KDF to create the encryption key. When using the password derivation function, select an appropriate number of iterations.

* No sample code due to conceptual rules.

If this is violated, the following may occur.

- If the password is smaller than the key, the full keyspace is not used. The remaining space is padded.
- Entropy is reduced by about a factor of four.

3.1.3.9 If using a password derivation function, select the appropriate number of iterations (Required)

When using a password derivation function, an appropriate number of iterations should be selected. For example, NIST recommends an iteration count of at least 10,000 for PBKDF2 and for critical keys where user-perceived performance is not critical at least 10,000,000. For critical keys, it is recommended to consider implementation of algorithms recognized by Password Hashing Competition (PHC) like Argon2.

* No sample code because of server-side rules.

If this is violated, the following may occur.

- A vulnerable key is generated.

3.1.3.10 Identify a standard implementation of the RNG algorithm that generates numbers with sufficient artificial randomness (Required)

Cryptographically secure RNGs generate random numbers that pass statistical randomness tests and are resistant to predictive attacks. Using random numbers generated by an RNG algorithm that does not meet the safe level increases the likelihood of a successful prediction attack. Therefore, it is necessary to privately use an RNG algorithm that generates numbers with sufficient artificial randomness.

Refer to the following for APIs that generate highly secure random numbers in the Android standard.

Rulebook

- *Use secure random number generator and settings (Required)*

If this is violated, the following may occur.

- Increased likelihood of a successful predictive attack.

3.1.3.11 All cryptography-related implementations properly manage memory state (Required)

All cryptographic implementations require that worker keys (like intermediate/derived keys in AES/DES/Rijndael) be properly removed from memory after consumption or in the event of an error. The internal state of the cipher also needs to be removed from memory as soon as possible.

AES Implementation and Post-Execution Deallocation:

```
package com.co.exsample.services

import java.security.SecureRandom
import javax.crypto.Cipher
import javax.crypto.SecretKey
import javax.crypto.spec.GCMParameterSpec
import javax.crypto.spec.SecretKeySpec

class AesGcmCipher {
    private val GCM_CIPHER_MODE = "AES/GCM/NoPadding" // Cipher mode (AEC GCM mode)
    private val GCM_NONCE_LENGTH = 12 // Nonce length

    private var key: SecretKey?
    private val tagBitLen: Int = 128
    private var aad: ByteArray?
    private val random = SecureRandom()

    constructor(key: ByteArray) {

        this.key = SecretKeySpec(key, "AES")

    }

    fun destroy() {
```

(continues on next page)

(continued from previous page)

```
        // release
        this.key = null
    }

    fun encrypt(plainData: ByteArray): ByteArray {

        val cipher = generateCipher(Cipher.ENCRYPT_MODE)
        val encryptData = cipher.doFinal(plainData)

        // Return nonce + Encrypt Data
        return cipher.iv + encryptData
    }

    fun decrypt(cipherData: ByteArray): ByteArray {
        val nonce = cipherData.copyOfRange(0, GCM_NONCE_LENGTH)
        val encryptData = cipherData.copyOfRange(GCM_NONCE_LENGTH, cipherData.size)

        val cipher = generateCipher(Cipher.DECRYPT_MODE, nonce)

        // Perform Decryption
        return cipher.doFinal(encryptData)
    }

    private fun generateCipher(mode: Int, nonceToDecrypt: ByteArray? = null):␣
↪Cipher {

        val cipher = Cipher.getInstance(GCM_CIPHER_MODE)

        // Get nonce
        val nonce = when (mode) {
            Cipher.ENCRYPT_MODE -> {
                // Generate nonce
                val nonceToEncrypt = ByteArray(GCM_NONCE_LENGTH)
                random.nextBytes(nonceToEncrypt)
                nonceToEncrypt
            }
            Cipher.DECRYPT_MODE -> {
                nonceToDecrypt ?: throw IllegalArgumentException()
            }
            else -> throw IllegalArgumentException()
        }

        // Create GCMParameterSpec
        val gcmParameterSpec = GCMParameterSpec(tagBitLen, nonce)

        cipher.init(mode, key, gcmParameterSpec)
        aad?.let {

            cipher.updateAAD(it)
        }

        return cipher
    }

    fun execute(text: String, keyBase64: String) {

        val key = Base64.getDecoder().decode(keyBase64)
        val cipher = AesGcmCipher(key)

        // encrypt
        val encryptData = cipher.encrypt(text.toByteArray())
```

(continues on next page)

(continued from previous page)

```
        // decrypt
        val decryptData = cipher.decrypt(encryptData)

        // release
        cipher.destroy()
    }
}
```

Rulebook

- *Key material must be erased from memory as soon as it is no longer needed (Required)*

If this is violated, the following may occur.

- Encrypted information left in memory is used in an unintended process.

3.1.3.12 Use industry-standard cryptographic APIs provided by the OS (Required)

Developing one' s own cryptographic functions is time consuming, difficult, and likely to fail. Instead, well-known algorithms that are widely recognized as secure can be used. Mobile operating systems provide standard cryptographic APIs that implement these algorithms and should be used for secure encryption.

For Android, encryption using Android KeyStore is recommended. See the rulebook below for sample code.

Data Storage and Privacy Requirements Rulebook

- *Encryption key storage method (Required)*

If this is violated, the following may occur.

- May result in an implementation containing vulnerabilities.

3.1.3.13 To counter padding oracle attacks, CBC should not be combined with HMAC or generate errors such as padding errors, MAC errors, decryption failures, etc. (Required)

In CBC mode, the plaintext block is XORed with the immediately preceding ciphertext block. This ensures that each encrypted block is unique and random, even if the blocks contain the same information.

Example of CBC mode implementation:

```
package com.co.exsample.services

import javax.crypto.Cipher
import javax.crypto.KeyGenerator
import javax.crypto.SecretKey

class CBCCipher {

    fun encrypt(text: String): Pair<ByteArray, ByteArray>{
        val plaintext: ByteArray = text.encodeToByteArray()
        val keygen = KeyGenerator.getInstance("AES")
        keygen.init(256)
        val key: SecretKey = keygen.generateKey()
        val cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING")
        cipher.init(Cipher.ENCRYPT_MODE, key)
        val ciphertextBuffer: ByteArray = cipher.doFinal(plaintext)
        val iv: ByteArray = cipher.iv

        return Pair(ciphertextBuffer, iv)
    }
}
```

If this is violated, the following may occur.

- Become vulnerable to padding oracle attacks.

3.1.3.14 When storing encrypted data, use a block mode such as Galois/Counter Mode (GCM) that also protects the integrity of the stored data (Recommended)

When storing encrypted data, it is recommended to use a block mode that also protects the integrity of the stored data, such as Galois/Counter Mode (GCM). The latter has the advantage that this algorithm is mandatory for each TLSv1.2 implementation and can therefore be used on all modern platforms.

Example of GCM mode implementation:

```
package com.co.exsample.services

import java.security.SecureRandom
import javax.crypto.Cipher
import javax.crypto.SecretKey
import javax.crypto.spec.GCMParameterSpec
import javax.crypto.spec.SecretKeySpec

class AesGcmCipher {
    private val GCM_CIPHER_MODE = "AES/GCM/NoPadding" // Cipher mode (AEC GCM mode)
    private val GCM_NONCE_LENGTH = 12 // Nonce length

    private var key: SecretKey?
    private val tagBitLen: Int = 128
    private var aad: ByteArray?
    private val random = SecureRandom()

    constructor(key: ByteArray) {

        this.key = SecretKeySpec(key, "AES")

    }

    fun destroy() {
        // release
        this.key = null
    }

    fun encrypt(plainData: ByteArray): ByteArray {

        val cipher = generateCipher(Cipher.ENCRYPT_MODE)
        val encryptData = cipher.doFinal(plainData)

        // Return nonce + Encrypt Data
        return cipher.iv + encryptData
    }

    fun decrypt(cipherData: ByteArray): ByteArray {
        val nonce = cipherData.copyOfRange(0, GCM_NONCE_LENGTH)
        val encryptData = cipherData.copyOfRange(GCM_NONCE_LENGTH, cipherData.size)

        val cipher = generateCipher(Cipher.DECRYPT_MODE, nonce)

        // Perform Decryption
        return cipher.doFinal(encryptData)
    }

    private fun generateCipher(mode: Int, nonceToDecrypt: ByteArray? = null):␣
↪Cipher {

        val cipher = Cipher.getInstance(GCM_CIPHER_MODE)
```

(continues on next page)

(continued from previous page)

```
        // Get nonce
        val nonce = when (mode) {
            Cipher.ENCRYPT_MODE -> {
                // Generate nonce
                val nonceToEncrypt = ByteArray(GCM_NONCE_LENGTH)
                random.nextBytes(nonceToEncrypt)
                nonceToEncrypt
            }
            Cipher.DECRYPT_MODE -> {
                nonceToDecrypt ?: throw IllegalArgumentException()
            }
            else -> throw IllegalArgumentException()
        }

        // Create GCMParameterSpec
        val gcmParameterSpec = GCMParameterSpec(tagBitLen, nonce)

        cipher.init(mode, key, gcmParameterSpec)
        aad?.let {

            cipher.updateAAD(it)
        }

        return cipher
    }

}
```

If this is violated, the following may occur.

- Easily identifiable patterns in the data.

3.1.3.15 IV is generated using a cryptographically secure random number generator (Required)

In CBC, OFB, CFB, PCBC, and GCM modes, the initialization vector (IV) is required as the initial input to the cipher. The IV need not be secret, but it must not be predictable. It must be random, unique, and non-reproducible for each encrypted message. Therefore, the IV must be generated using a cryptographically secure random number generator. For more information on IVs, see Crypto Fail' s article on initialization vectors.

See the rulebook below for sample code.

Rulebook

- *Use secure random number generator and settings (Required)*

If this is violated, the following may occur.

- A predictable initialization vector is generated.

3.1.3.16 Note that IV is used differently when using CTR and GCM modes, where the initialization vector is often a counter (Required)

Note that IVs are used differently when using CTR and GCM modes, where the initialization vector is often a counter (a combination of CTR and nonce). Therefore, it is necessary to use a predictable IV with its own stateful model. The CTR uses a new nonce and counter as input for each new block operation. Example : In the case of a plaintext of 5120 bits in length, there are 20 blocks, so 20 input vectors consisting of a nonce and a counter are needed. GCM, on the other hand, has only one IV per encryption operation and does not repeat with the same key. For details and recommendations on IVs, see section 8 of NIST document on GCM.

* No sample code due to conceptual rules.

If this is violated, the following may occur.

- Failure to meet the initialization vector requirements for each mode.

3.1.3.17 Use OAEP incorporated in PKCS#1 v2.0 as a padding mechanism for asymmetric encryption (Required)

Previously, PKCS1.5 padding (code: PKCS1Padding) was used as a padding mechanism for asymmetric encryption. This mechanism is vulnerable to the padding Oracle attack. Therefore, PKCS#1 v2.0 (codes: OAEPwithSHA-256andMGF1Padding , OAEPwithSHA-224andMGF1Padding , and OAEPwithSHA-384andMGF1Padding , OAEPwithSHA-512andMGF1Padding). OAEP is the most appropriate method to use. Note that even if you use OAEP, you may encounter the well-known problem known as the Mangers attack described in Kudelskisecurity' s blog.

The sample code below shows how OAEP is used.

```
val key: Key = ...
val cipher = Cipher.getInstance("RSA/ECB/OAEPPadding")
        .apply {
            // To use SHA-256 the main digest and SHA-1 as the MGF1 digest
            init(Cipher.ENCRYPT_MODE, key, OAEPParameterSpec("SHA-256", "MGF1",␣
↪MGF1ParameterSpec.SHA1, PSource.PSpecified.DEFAULT))
            // To use SHA-256 for both digests
            init(Cipher.ENCRYPT_MODE, key, OAEPParameterSpec("SHA-256", "MGF1",␣
↪MGF1ParameterSpec.SHA256, PSource.PSpecified.DEFAULT))
        }
```

If this is violated, the following may occur.

- Become vulnerable to padding oracle attacks.

3.1.3.18 Use key considering memory dump (Required)

If memory dumps are part of the threat model, the key can be accessed at the moment the key is actively used. Memory dumps require root access (e.g. rooted or jailbroken devices) or an application patched with Frida (so that tools such as Fridump can be used). Therefore, if a key is still needed on a device, it is best to consider the following

- Keys on remote servers: remote key vaults such as Amazon KMS or Azure Key Vault can be used. For some use cases, developing an orchestration layer between the app and the remote resource may be an appropriate option.
- Keys in hardware-protected secure storage: all cryptographic actions and the keys themselves must be in a trusted execution environment (e.g., Android Keystore. For more information, see Android Data Storage For more information.
- Keys protected by envelope encryption: If keys are stored outside of TEE/SE, consider using multi-layered encryption. Envelope encryption approach ([OWASP Cryptographic Storage Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_ Sheet.html#encrypting-stored-keys), Google Cloud Key management guide, and AWS Well-Architected Framework guide See also](https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-08), or the HPKE approach to key encrypt data encryption keys.
- Keys in memory: Ensure that keys remain in memory for as little time as possible, and consider zeroing and deactivating keys after a successful encryption operation or in the event of an error. For general encryption guidelines, see Erasing Memory of Sensitive Data. For more detailed information, see "Testing Memory for Sensitive Data" . The sample code below is a process for preventing the leakage of keys in memory in an application.

```
val secret: ByteArray? = null
try {
    //get or generate the secret, do work with it, make sure you make no local␣
↪copies
} finally {
    if (null != secret) {
        Arrays.fill(secret, 0.toByte())
```

(continues on next page)

(continued from previous page)

```
    }
}
```

If this is violated, the following may occur.

- Keys in memory can be leaked.

3.1.3.19 Do not share the same key across accounts or devices (Required)

To facilitate memory dumps, the same key is not shared among accounts or devices, except for the public key used for signature verification and encryption.

* No sample code due to deprecated rules.

If this is violated, the following may occur.

- Facilitate a memory dump of the key.

3.1.3.20 Appropriate key protection by means of transport symmetric keys or other mechanisms (Required)

If keys need to be transferred between devices or from the app to the backend, ensure that proper key protection is in place through transport symmetric keys or other mechanisms. In many cases, keys are shared in an obfuscated state and can be easily undone. Instead, ensure that asymmetric encryption or wrapping keys are used. For example, a symmetric key can be encrypted with an asymmetric public key.

KeyStore is used to properly protect keys. See the rulebook below for key storage with KeyStore.

Rulebook

- *Verify that keys are stored inside security hardware (Recommended)*

If this is violated, the following may occur.

- The key is undone and read.

3.1.3.21 Do not use hardcoded symmetric encryption as the only method of encryption (Required)

Do not use hardcoded symmetric encryption as the only method of encryption. The following is an example of a check procedure.

1. identify all instances of symmetric encryption
2. check for the presence of a hardcoded symmetric key for each instance identified
3. check if hard-coded symmetric encryption is not used as the only method of encryption

* No sample code for debugging methods.

If this is violated, the following may occur.

- The encryption scheme is read.

3.1.3.22 Perform a method trace of the encryption method to verify the write destination or read source of the key material (Required)

Perform a method trace of the encryption method of the encryption method to determine the input/output values of the keys and other data being used. Monitor file system accesses during the execution of encryption operations to evaluate the write destination or read source of key material. For example, RMS - Runtime Mobile Security, API Monitor can be used to monitor the file system.

If this is violated, the following may occur.

- The key material is used in a process for which it was not intended.

3.2 MSTG-CRYPTO-2

The app uses proven implementations of cryptographic primitives.

3.2.1 Problematic Encryption Configuration

* Check the contents of MSTG-CRYPTO-1 3.1.1. Problematic Encryption Configuration.

3.2.2 Configuration of Cryptographic Standard Algorithms

These test cases focus on implementation and use of cryptographic primitives. Following checks should be performed:

- identify all instance of cryptography primitives and their implementation (library or custom implementation)
- verify how cryptography primitives are used and how they are configured
- verify if cryptographic protocols and algorithms used are not deprecated for security purposes.

Reference

- owasp-mastg Testing the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4) Overview

RuleBook

- *Configure the appropriate encryption standard algorithm (Required)*

3.2.2.1 Static Analysis

Identify all the instances of the cryptographic primitives in code. Identify all custom cryptography implementations. You can look for:

- classes Cipher, Mac, MessageDigest, Signature
- interfaces Key, PrivateKey, PublicKey, SecretKey
- functions getInstance, generateKey
- exceptions KeyStoreException, CertificateException, NoSuchAlgorithmException
- classes which uses java.security.*, javax.crypto.*, android.security.* and android.security.keystore.* packages.

Identify that all calls to getInstance use default provider of security services by not specifying it (it means AndroidOpenSSL aka Conscrypt). Provider can only be specified in KeyStore related code (in that situation KeyStore should be provided as provider). If other provider is specified it should be verified according to situation and business case (i.e. Android API version), and provider should be examined against potential vulnerabilities.

Ensure that the best practices outlined in the "Cryptography for Mobile Apps" chapter are followed. Look at insecure and deprecated algorithms and common configuration issues.

Reference

- owasp-mastg Testing the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4) Static Analysis

RuleBook

- *Identifies all calls to getInstance that do not specify a provider of security services (Required)*

3.2.2.2 Dynamic Analysis

* Omitted in this chapter because the same information is provided in MSTG-CRYPTO-1 .

Reference

- owasp-mastg Testing the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4) Dynamic Analysis

3.2.3 Rulebook

1. *Configure the appropriate encryption standard algorithm (Required)*
2. *Identifies all calls to getInstance that do not specify a provider of security services (Required)*

3.2.3.1 Configure the appropriate encryption standard algorithm (Required)

All calls to getInstance on instances of cryptographic primitives use the default provider of the security service without specifying it (meaning AndroidOpenSSL, aka Conscrypt).Providers can only be specified in KeyStore-related code (in which case KeyStore must be provided as a provider).If other providers are specified, they should be validated according to the situation and business case (Android API Version) , and the provider should be examined for potential vulnerabilities.

The following is an example of keywords related to encryption primitives.

- Class : Cipher , Mac , MessageDigest , Signature
- Interfaces : Key , PrivateKey , PublicKey , SecretKey
- Functions : getInstance , generateKey
- Exceptions: KeyStoreException , CertificateException , NoSuchAlgorithmException
- java.security.* , javax.crypto.* , android.security.* , android.security.keystore.*

Also ensure that it is not deprecated for security reasons.

If this is violated, the following may occur.

- Encryption algorithms containing potential vulnerabilities are used.

3.2.3.2 Identifies all calls to getInstance that do not specify a provider of security services (Required)

Identifies all calls to getInstance that do not specify a provider of security services. The following methods should be used to find out the provider.

```
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(new KeyStore.LoadStoreParameter() {
    @Override
    public KeyStore.ProtectionParameter getProtectionParameter() {
        return null;
    }
});

Provider provider = keyStore.getProvider();
```

If this is violated, the following may occur.

- Encryption algorithms containing potential vulnerabilities are used.

3.3 MSTG-CRYPTO-3

The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.

3.3.1 Problematic Encryption Configuration

* Check the contents of MSTG-CRYPTO-1 3.1.1. Problematic Encryption Configuration.

3.3.2 Configuration of Cryptographic Standard Algorithms

* Check the contents of MSTG-CRYPTO-2 3.2.2. Configuration of Cryptographic Standard Algorithms.

3.4 MSTG-CRYPTO-4

The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.

3.4.1 Insecure or deprecated encryption algorithms

When assessing a mobile app, you should make sure that it does not use cryptographic algorithms and protocols that have significant known weaknesses or are otherwise insufficient for modern security requirements. Algorithms that were considered secure in the past may become insecure over time; therefore, it's important to periodically check current best practices and adjust configurations accordingly.

Verify that cryptographic algorithms are up to date and in-line with industry standards. Vulnerable algorithms include outdated block ciphers (such as DES and 3DES), stream ciphers (such as RC4), hash functions (such as MD5 and SHA1), and broken random number generators (such as Dual_EC_DRBG and SHA1PRNG). Note that even algorithms that are certified (for example, by NIST) can become insecure over time. A certification does not replace periodic verification of an algorithm's soundness. Algorithms with known weaknesses should be replaced with more secure alternatives. Additionally, algorithms used for encryption must be standardized and open to verification. Encrypting data using any unknown, or proprietary algorithms may expose the application to different cryptographic attacks which may result in recovery of the plaintext.

Inspect the app's source code to identify instances of cryptographic algorithms that are known to be weak, such as:

- DES, 3DES
- RC2
- RC4
- BLOWFISH
- MD4
- MD5
- SHA1

The names of cryptographic APIs depend on the particular mobile platform.

Please make sure that:

- Cryptographic algorithms are up to date and in-line with industry standards. This includes, but is not limited to outdated block ciphers (e.g. DES), stream ciphers (e.g. RC4), as well as hash functions (e.g. MD5) and broken random number generators like Dual_EC_DRBG (even if they are NIST certified). All of these should be marked as insecure and should not be used and removed from the application and server.
- Key lengths are in-line with industry standards and provide protection for sufficient amount of time. A comparison of different key lengths and protection they provide taking into account Moore' s law is available online.
- Cryptographic means are not mixed with each other: e.g. you do not sign with a public key, or try to reuse a key pair used for a signature to do encryption.
- Cryptographic parameters are well defined within reasonable range. This includes, but is not limited to: cryptographic salt, which should be at least the same length as hash function output, reasonable choice of password derivation function and iteration count (e.g. PBKDF2, scrypt or bcrypt), IVs being random and unique, fit-for-purpose block encryption modes (e.g. ECB should not be used, except specific cases), key management being done properly (e.g. 3DES should have three independent keys) and so on.

Reference

- owasp-mastg Identifying Insecure and/or Deprecated Cryptographic Algorithms (MSTG-CRYPTO-4)

Rulebook

- *Do not use unsecured or deprecated encryption algorithms (Required)*

3.4.2 Configuration of Cryptographic Standard Algorithms

* Check the contents of MSTG-CRYPTO-2 3.2.2. Configuration of Cryptographic Standard Algorithms.

3.4.3 Rulebook

1. *Do not use unsecured or deprecated encryption algorithms (Required)*

3.4.3.1 Do not use unsecured or deprecated encryption algorithms (Required)

Implement with the latest encryption algorithms compliant with industry standards.

Specific aspects of implementation should be in accordance with the following contents.

- The encryption algorithm is up-to-date and conforms to industry standards. See "*Set key lengths that meet industry standards (Required)*" for industry standards.
- Key lengths comply with industry standards and provide sufficient time protection. A comparison of various key lengths and their protection performance considering Moore' s Law can be found online.
- Do not mix encryption methods with each other: for example, do not sign with the public key or reuse the symmetric key used for signing for encryption.
- Define cryptographic parameters as reasonably appropriate. These include cipher salt that must be at least as long as the hash function output, appropriate choice of password derivation function and number of iterations (e.g. PBKDF2 , scrypt, bcrypt), IV must be random and unique, block cipher mode appropriate for the purpose (e.g. ECB should not be used except in certain cases), and key management must be appropriate (e.g. 3DES should have three independent keys).), that the IV is random and unique, that the block encryption mode is appropriate for the purpose (e.g. ECB should not be used except in certain cases), and that key management is properly implemented (e.g. 3DES should have three independent keys).

See the rulebook below for sample code.

Rulebook

- *Use encryption for your purposes (Required)*

If this is violated, the following may occur.

- May result in a weak encryption process.

3.5 MSTG-CRYPTO-5

The app doesn't re-use the same cryptographic key for multiple purposes.

3.5.1 Verification of the intended use and reuse of encryption keys

This test case focuses on verification of purpose and reusage of the same cryptographic keys. The following checks should be performed:

- identify all instances where cryptography is used
- identify the purpose of the cryptographic material (to protect data in use, in transit or at rest)
- identify type of cryptography
- verify if cryptography is used according to its purpose

3.5.1.1 Static Analysis

Identify all instances where cryptography is used. You can look for:

- classes Cipher, Mac, MessageDigest, Signature
- interfaces Key, PrivateKey, PublicKey, SecretKey
- functions getInstance, generateKey
- exceptions KeyStoreException, CertificateException, NoSuchAlgorithmException
- classes importing java.security.*, javax.crypto.*, android.security.*, android.security.keystore.*

For each identified instance, identify its purpose and its type. It can be used:

- for encryption/decryption - to ensure data confidentiality
- for signing/verifying - to ensure integrity of data (as well as accountability in some cases)
- for maintenance - to protect keys during certain sensitive operations (such as being imported to the KeyStore)

Additionally, you should identify the business logic which uses identified instances of cryptography.

During verification the following checks should be performed:

- are all keys used according to the purpose defined during its creation? (it is relevant to KeyStore keys, which can have KeyProperties defined)
- for asymmetric keys, is the private key being exclusively used for signing and the public key encryption?
- are symmetric keys used for multiple purposes? A new symmetric key should be generated if it's used in a different context.
- is cryptography used according to its business purpose?

Reference

- owasp-mastg Testing the Purposes of Keys (MSTG-CRYPTO-5) Static Analysis

RuleBook

- *Use encryption for your purposes (Required)*

3.5.1.2 Dynamic Analysis

* Check the contents of MSTG-CRYPTO-1 3.1.2.2. Dynamic Analysis .

Reference

- owasp-mastg Testing the Purposes of Keys (MSTG-CRYPTO-5) Dynamic Analysis

3.5.2 Rulebook

1. *Use encryption for your purposes (Required)*

3.5.2.1 Use encryption for your purposes (Required)

Use encryption appropriate for the purpose.

The following specific aspects must be complied with: * All keys must be defined at the time of creation.

- All keys are used according to the purpose defined at the time of creation (related to the KeyStore keys for which KeyProperties can be defined).
- For asymmetric keys, the private key is used for signing and the public key is used for encryption only.
- Symmetric keys are not used for multiple purposes. If used in different contexts, a new symmetric key must be generated.
- Encryption should be used for business purposes.

The following is an example of keywords related to encryption in the source code.

- Class : Cipher , Mac , MessageDigest , Signature
- Interfaces : Key , PrivateKey , PublicKey , SecretKey
- Functions : getInstance , generateKey
- Exceptions: KeyStoreException , CertificateException , NoSuchAlgorithmException
- java.security.* , javax.crypto.* , android.security.* , android.security.keystore.*

If this is violated, the following may occur.

- May result in a weak encryption process.

3.6 MSTG-CRYPTO-6

All random values are generated using a sufficiently secure random number generator.

3.6.1 Random Number Generator Selection

This test case focuses on random values used by application. The following checks should be performed:

- identify all instances where random values are used
- verify if random number generators are not considered as being cryptographically secure
- verify how random number generators are used
- verify randomness of the generated random values

Identify all the instances of random number generators and look for either custom or well-known insecure classes. For instance, java.util.Random produces an identical sequence of numbers for each given seed value; consequently, the sequence of numbers is predictable. Instead a well-vetted algorithm should be chosen that is currently considered to be strong by experts in the field, and a well-tested implementations with adequate length seeds should be used.

Reference

- owasp-mastg Testing Random Number Generation (MSTG-CRYPTO-6) Overview

RuleBook

- *Use secure random number generator and settings (Required)*

3.6.1.1 Secure random number generator

Identify all instances of SecureRandom that are not created using the default constructor. Specifying the seed value may reduce randomness. Prefer the no-argument constructor of SecureRandom that uses the system-specified seed value to generate a 128-byte-long random number.

Reference

- owasp-mastg Testing Random Number Generation (MSTG-CRYPTO-6) Static Analysis

RuleBook

- *Use secure random number generator and settings (Required)*

3.6.1.2 Unsecured random number generator

In general, if a PRNG is not advertised as being cryptographically secure (e.g. java.util.Random), then it is probably a statistical PRNG and should not be used in security-sensitive contexts. Pseudo-random number generators can produce predictable numbers if the generator is known and the seed can be guessed. A 128-bit seed is a good starting point for producing a "random enough" number.

Once an attacker knows what type of weak pseudo-random number generator (PRNG) is used, it can be trivial to write a proof-of-concept to generate the next random value based on previously observed ones, as it was done for Java Random. In case of very weak custom random generators it may be possible to observe the pattern statistically. Although the recommended approach would anyway be to decompile the APK and inspect the algorithm (see Static Analysis).

If you want to test for randomness, you can try to capture a large set of numbers and check with the Burp' s sequencer to see how good the quality of the randomness is.

You can use method tracing on the mentioned classes and methods to determine input / output values being used.

Reference

- owasp-mastg Testing Random Number Generation (MSTG-CRYPTO-6) Static Analysis
- owasp-mastg Testing Random Number Generation (MSTG-CRYPTO-6) Dynamic Analysis

RuleBook

- *Use secure random number generator and settings (Required)*

3.6.2 Rulebook

1. *Use secure random number generator and settings (Required)*

3.6.2.1 Use secure random number generator and settings (Required)

When using random numbers, security depends on the randomness of the random numbers. To increase security, it is necessary to generate random numbers using a more secure random number generator and configuration.

The following is an example of a secure random number generator.

- SecureRandom

When using SecureRandom, the constructor with no arguments creates an Generate an instance. If an instance is created by specifying a seed value, the randomness may be reduced. Therefore, the seed value specified by the system is used to generate a random number 128 bytes long.

The following source code is an example of SecureRandom instance generation by a constructor with no arguments.

```
import java.security.SecureRandom;
// ...

public static void main (String args[]) {
  SecureRandom number = new SecureRandom();
  // Generate 20 integers 0..20
  for (int i = 0; i < 20; i++) {
    System.out.println(number.nextInt(21));
  }
}
```

On the other hand, non-secure random number generators include

- java.util.Random

The above are pseudo-random number generators (PRNGs) and should not be used in security-sensitive contexts. Pseudo-random number generators can produce predictable numbers if the generator is known and the seed value can be guessed.

The following source code is an example of random number generation by java.util.Random.

```
import java.util.Random;
// ...

Random number = new Random(123L);
//...
for (int i = 0; i < 20; i++) {
  // Generate another random integer in the range [0, 20]
  int n = number.nextInt(21);
  System.out.println(n);
}
```

If this is violated, the following may occur.

- Insecure random numbers are generated.

4 Authentication and Session Management Requirements

4.1 MSTG-AUTH-1

If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint.

* Refer to "*1.2.1.1. Appropriate authentication response in MSTG-ARCH-2*" for appropriate authentication support in this chapter.

Reference

- owasp-mastg Verifying that Appropriate Authentication is in Place (MSTG-ARCH-2 and MSTG-AUTH-1)
- owasp-mastg Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3)

4.1.1 Testing Confirm Credentials

The confirm credential flow is available since Android 6.0 and is used to ensure that users do not have to enter app-specific passwords together with the lock screen protection. Instead: if a user has logged in to the device recently, then confirm-credentials can be used to unlock cryptographic materials from the AndroidKeystore. That is, if the user unlocked the device within the set time limits (setUserAuthenticationValidityDurationSeconds), otherwise the device needs to be unlocked again.

Note that the security of Confirm Credentials is only as strong as the protection set at the lock screen. This often means that simple predictive lock-screen patterns are used and therefore we do not recommend any apps which require L2 of security controls to use Confirm Credentials.Note that setUserAuthenticationValidityDurationSeconds is deprecated in API level 30, so setUserAuthenticationParameters is currently recommended.

Reference

- owasp-mastg Testing Confirm Credentials (MSTG-AUTH-1 and MSTG-STORAGE-11) Overview

Rulebook

- *Credentials verification flow should be used with an understanding of security strength (Recommended)*

4.1.1.1 Static Analysis

Reassure that the lock screen is set:

```
KeyguardManager mKeyguardManager = (KeyguardManager) getSystemService(Context.
↪KEYGUARD_SERVICE);
if (!mKeyguardManager.isKeyguardSecure()) {
    // Show a message that the user hasn't set up a lock screen.
}
```

- Create the key protected by the lock screen. In order to use this key, the user needs to have unlocked the device in the last X seconds, or the device needs to be unlocked again. Make sure that this timeout is not too long, as it becomes harder to ensure that it was the same user using the app as the user unlocking the device:

```
try {
    KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
    keyStore.load(null);
    KeyGenerator keyGenerator = KeyGenerator.getInstance(
            KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

    // Set the alias of the entry in Android KeyStore where the key will␣
↪appear
    // and the constrains (purposes) in the constructor of the Builder
    keyGenerator.init(new KeyGenParameterSpec.Builder(KEY_NAME,
            KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
            .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
            .setUserAuthenticationRequired(true)
                    // Require that the user has unlocked in the last 30␣
↪seconds
            .setUserAuthenticationValidityDurationSeconds(30)
            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
            .build());
    keyGenerator.generateKey();
} catch (NoSuchAlgorithmException | NoSuchProviderException
        | InvalidAlgorithmParameterException | KeyStoreException
        | CertificateException | IOException e) {
    throw new RuntimeException("Failed to create a symmetric key", e);
}
```

- Setup the lock screen to confirm:

```
private static final int REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS = 1; //used␣
↪as a number to verify whether this is where the activity results from
Intent intent = mKeyguardManager.createConfirmDeviceCredentialIntent(null,␣
↪null);
if (intent != null) {
    startActivityForResult(intent, REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS);
}
```

Note that startActivityForResult is currently deprecated; see the rulebook for recommended methods.

- Use the key after lock screen:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data)
↪{
    if (requestCode == REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS) {
        // Challenge completed, proceed with using cipher
        if (resultCode == RESULT_OK) {
            //use the key for the actual authentication flow
        } else {
            // The user canceled or didn't complete the lock screen
            // operation. Go to error/cancellation flow.
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

Note that onActivityResult is currently deprecated; see the rulebook for recommended methods.

Make sure that the unlocked key is used during the application flow. For example, the key may be used to decrypt local storage or a message received from a remote endpoint. If the application simply checks whether the user has unlocked the key or not, the application may be vulnerable to a local authentication bypass.

Reference

- owasp-mastg Testing Confirm Credentials (MSTG-AUTH-1 and MSTG-STORAGE-11) Static Analysis

Rulebook

- *Credentials verification flow should be used with an understanding of security strength (Recommended)*

4.1.1.2 Dynamic Analysis

Validate the duration of time (seconds) for which the key is authorized to be used after the user is successfully authenticated. This is only needed if setUserAuthenticationRequired is used.

Reference

- owasp-mastg Testing Confirm Credentials (MSTG-AUTH-1 and MSTG-STORAGE-11) Dynamic Analysis

Rulebook

- *Credentials verification flow should be used with an understanding of security strength (Recommended)*

4.1.2 Rulebook

1. *Credentials verification flow should be used with an understanding of security strength (Recommended)*

4.1.2.1 Credentials verification flow should be used with an understanding of security strength (Recommended)

Note that the security of Confirm Credentials is only as strong as the protection set at the lock screen. This often means that simple predictive lock-screen patterns are used and therefore we do not recommend any apps which require L2 of security controls to use Confirm Credentials.

Below is a sample code of the credential verification flow.

```
    private void createNewKey(String alias, int timeout) {
        try {
            KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
            keyStore.load(null);
            KeyGenerator keyGenerator = KeyGenerator.getInstance(
                    KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

            // Set the alias of the entry in Android KeyStore where the key will␣
↪appear
            // and the constrains (purposes) in the constructor of the Builder
            KeyGenParameterSpec.Builder builder = new KeyGenParameterSpec.
↪Builder(alias,
                    KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
                    .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
                    .setUserAuthenticationRequired(true)
                    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7);
            // Require that the user has unlocked in the last 30 seconds
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
```

(continues on next page)

(continued from previous page)

```
                builder.setUserAuthenticationParameters(timeout, KeyProperties.
↪AUTH_DEVICE_CREDENTIAL);
            } else {
                builder.setUserAuthenticationValidityDurationSeconds(timeout);
            }
            keyGenerator.init(builder.build());
            keyGenerator.generateKey();
        } catch (NoSuchAlgorithmException | NoSuchProviderException
                | InvalidAlgorithmParameterException | KeyStoreException
                | CertificateException | IOException e) {
            throw new RuntimeException("Failed to create a symmetric key", e);
        }
    }

    private ActivityResultLauncher<Intent> checkCredentialLauncher =␣
↪registerForActivityResult(new ActivityResultContracts.StartActivityForResult(),
        result -> {
            if (result.getResultCode() == RESULT_OK) {
                // use the key for the actual authentication flow
            } else {
                // The user canceled or didn't complete the lock screen
                // operation. Go to error/cancellation flow.
            }
        });

    private void checkCredential(Context context) {
        KeyguardManager keyguardManager = (KeyguardManager)context.
↪getSystemService(Context.KEYGUARD_SERVICE);
        // Terminal authentication validity judgment
        if (keyguardManager.isKeyguardSecure()) {
            // Judgment of OS10 or higher
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
                BiometricPrompt biometricPrompt;
                // Judgment of OS11 or higher
                if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
                    biometricPrompt = new BiometricPrompt.Builder(context)
                            .setAllowedAuthenticators(BiometricManager.
↪Authenticators.DEVICE_CREDENTIAL)
                            .build();
                } else {
                    biometricPrompt = new BiometricPrompt.Builder(context)
                            .setDeviceCredentialAllowed(true)
                            .build();
                }

                CancellationSignal cancellationSignal = new CancellationSignal();
                cancellationSignal.setOnCancelListener(() -> {
                    // When requesting cancellation of authorization
                });

                Executor executors = ContextCompat.getMainExecutor(context);
                BiometricPrompt.AuthenticationCallback authCallBack = new␣
↪BiometricPrompt.AuthenticationCallback() {
                    @Override
                    public void onAuthenticationError(int errorCode, CharSequence␣
↪errString) {
                        super.onAuthenticationError(errorCode, errString);
                        // Authentication Error
                    }

                    @Override
```

(continues on next page)

(continued from previous page)

```
                    public void onAuthenticationSucceeded(BiometricPrompt.
↪AuthenticationResult result) {
                        super.onAuthenticationSucceeded(result);
                        // Authentication Success
                    }

                    @Override
                    public void onAuthenticationFailed() {
                        super.onAuthenticationFailed();
                        // Authentication failure
                    }
                };
                biometricPrompt.authenticate(cancellationSignal, executors,␣
↪authCallBack);
            } else {
                Intent intent = keyguardManager.
↪createConfirmDeviceCredentialIntent(null, null);
                checkCredentialLauncher.launch(intent);
            }
        }
    }
```

If this is not noted, the following may occur.

- Potentially enforce a low-strength security credential verification flow.

4.2 MSTG-AUTH-2

If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user' s credentials.

4.2.1 Session Information Management

Stateful (or "session-based") authentication is characterized by authentication records on both the client and server. The authentication flow is as follows:

1. The app sends a request with the user' s credentials to the backend server.
2. The server verifies the credentials If the credentials are valid, the server creates a new session along with a random session ID.
3. The server sends to the client a response that includes the session ID.
4. The client sends the session ID with all subsequent requests. The server validates the session ID and retrieves the associated session record.
5. After the user logs out, the server-side session record is destroyed and the client discards the session ID.

When sessions are improperly managed, they are vulnerable to a variety of attacks that may compromise the session of a legitimate user, allowing the attacker to impersonate the user. This may result in lost data, compromised confidentiality, and illegitimate actions.

Reference

- owasp-mastg Testing Stateful Session Management (MSTG-AUTH-2)

4.2.1.1 Session Management Best Practices

Locate any server-side endpoints that provide sensitive information or functions and verify the consistent enforcement of authorization. The backend service must verify the user's session ID or token and make sure that the user has sufficient privileges to access the resource. If the session ID or token is missing or invalid, the request must be rejected.

Make sure that:

- Session IDs are randomly generated on the server side.
- The IDs can't be guessed easily (use proper length and entropy).
- Session IDs are always exchanged over secure connections (e.g. HTTPS).
- The mobile app doesn't save session IDs in permanent storage.
- The server verifies the session whenever a user tries to access privileged application elements, (a session ID must be valid and must correspond to the proper authorization level).
- The session is terminated on the server side and session information deleted within the mobile app after it times out or the user logs out.

Authentication shouldn't be implemented from scratch but built on top of proven frameworks. Many popular frameworks provide ready-made authentication and session management functionality. If the app uses framework APIs for authentication, check the framework security documentation for best practices. Security guides for common frameworks are available at the following links:

- Spring (Java)
- Struts (Java)
- Laravel (PHP)
- Ruby on Rails

A great resource for testing server-side authentication is the OWASP Web Testing Guide, specifically the Testing Authentication and Testing Session Management chapters.

Reference

- owasp-mastg Testing Stateful Session Management (MSTG-AUTH-2) Session Management Best Practices

Rulebook

- *Ensure mobile apps do not store session IDs in persistent storage (Required)*

4.2.2 Rulebook

1. *Ensure mobile apps do not store session IDs in persistent storage (Required)*

4.2.2.1 Ensure mobile apps do not store session IDs in persistent storage (Required)

Storing session IDs in persistent storage may be read/written by users or used by third parties. Therefore, session IDs should not be stored in such storage.

* No sample code due to deprecated rules.

If this is violated, the following may occur.

- There is a risk that the session ID can be read/written by the user or used by a third party.

4.3 MSTG-AUTH-3

If stateless token-based authentication is used, the server provides a token that has been signed using a secure algorithm.

4.3.1 Token management

Token-based authentication is implemented by sending a signed token (verified by the server) with each HTTP request. The most commonly used token format is the JSON Web Token, defined in RFC7519. A JWT may encode the complete session state as a JSON object. Therefore, the server doesn't have to store any session data or authentication information.

JWT tokens consist of three Base64Url-encoded parts separated by dots. The Token structure is as follows:

```
base64UrlEncode(header).base64UrlEncode(payload).base64UrlEncode(signature)
```

The following example shows a Base64Url-encoded JSON Web Token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

The header typically consists of two parts: the token type, which is JWT, and the hashing algorithm being used to compute the signature. In the example above, the header decodes as follows:

```
{"alg":"HS256","typ":"JWT"}
```

The second part of the token is the payload, which contains so-called claims. Claims are statements about an entity (typically, the user) and additional metadata. For example:

```
{"sub":"1234567890","name":"John Doe","admin":true}
```

The signature is created by applying the algorithm specified in the JWT header to the encoded header, encoded payload, and a secret value. For example, when using the HMAC SHA256 algorithm the signature is created in the following way:

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

Note that the secret is shared between the authentication server and the backend service - the client does not know it. This proves that the token was obtained from a legitimate authentication service. It also prevents the client from tampering with the claims contained in the token.

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3)

4.3.2 Static Analysis

Identify the JWT library that the server and client use. Find out whether the JWT libraries in use have any known vulnerabilities.

Verify that the implementation adheres to JWT best practices:

- Verify that the HMAC is checked for all incoming requests containing a token;
- Verify the location of the private signing key or HMAC secret key. The key should remain on the server and should never be shared with the client. It should be available for the issuer and verifier only.
- Verify that no sensitive data, such as personal identifiable information, is embedded in the JWT. If, for some reason, the architecture requires transmission of such information in the token, make sure that payload encryption is being applied. See the sample Java implementation on the OWASP JWT Cheat Sheet.

- Make sure that replay attacks are addressed with the jti (JWT ID) claim, which gives the JWT a unique identifier.
- Make sure that cross service relay attacks are addressed with the aud (audience) claim, which defines for which application the token is entitled.
- Verify that tokens are stored securely on the mobile phone, with, for example, KeyStore (Android).

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Static Analysis

Rulebook

- *Identify JWT libraries in use and whether they have known vulnerabilities (Required)*
- *Ensure tokens are securely stored on mobile devices by KeyStore (Android) (Required)*

4.3.2.1 Enforcing the Hashing Algorithm

An attacker executes this by altering the token and, using the 'none' keyword, changing the signing algorithm to indicate that the integrity of the token has already been verified. Some libraries might treat tokens signed with the 'none' algorithm as if they were valid tokens with verified signatures, so the application will trust altered token claims.

For example, in Java applications, the expected algorithm should be requested explicitly when creating the verification context:

```
// HMAC key - Block serialization and storage as String in JVM memory
private transient byte[] keyHMAC = ...;

//Create a verification context for the token requesting explicitly the use of the␣
↪HMAC-256 HMAC generation

JWTVerifier verifier = JWT.require(Algorithm.HMAC256(keyHMAC)).build();

//Verify the token; if the verification fails then an exception is thrown

DecodedJWT decodedToken = verifier.verify(token);
```

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Enforcing the Hashing Algorithm

Rulebook

- *Need to request the expected hash algorithm (Required)*

4.3.2.2 Token Expiration

Once signed, a stateless authentication token is valid forever unless the signing key changes. A common way to limit token validity is to set an expiration date. Make sure that the tokens include an "exp" expiration claim and the backend doesn't process expired tokens.

A common method of granting tokens combines access tokens and refresh tokens. When the user logs in, the backend service issues a short-lived access token and a long-lived refresh token. The application can then use the refresh token to obtain a new access token, if the access token expires.

For apps that handle sensitive data, make sure that the refresh token expires after a reasonable period of time. The following example code shows a refresh token API that checks the refresh token's issue date. If the token is not older than 14 days, a new access token is issued. Otherwise, access is denied and the user is prompted to login again.

```
app.post('/renew_access_token', function (req, res) {
    // verify the existing refresh token
    var profile = jwt.verify(req.body.token, secret);
```

(continues on next page)

(continued from previous page)

```
    // if refresh token is more than 14 days old, force login
    if (profile.original_iat - new Date() > 14) { // iat == issued at
      return res.send(401); // re-login
    }

    // check if the user still exists or if authorization hasn't been revoked
    if (!valid) return res.send(401); // re-logging

    // issue a new access token
    var renewed_access_token = jwt.sign(profile, secret, { expiresInMinutes: 60*5 }
↪);
    res.json({ token: renewed_access_token });
  });
```

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Token Expiration

Rulebook

- *Set an appropriate period for refresh token expiration (Required)*

4.3.3 Dynamic Analysis

Investigate the following JWT vulnerabilities while performing dynamic analysis:

- Token Storage on the client:
  - The token storage location should be verified for mobile apps that use JWT.
- Cracking the signing key:
  - Token signatures are created via a private key on the server. After you obtain a JWT, choose a tool for brute forcing the secret key offline.
- Information Disclosure:
  - Decode the Base64Url-encoded JWT and find out what kind of data it transmits and whether that data is encrypted.
- Tampering with the Hashing Algorithm:
  - Usage of asymmetric algorithms. JWT offers several asymmetric algorithms as RSA or ECDSA. When these algorithms are used, tokens are signed with the private key and the public key is used for verification. If a server is expecting a token to be signed with an asymmetric algorithm and receives a token signed with HMAC, it will treat the public key as an HMAC secret key. The public key can then be misused, employed as an HMAC secret key to sign the tokens.
  - Modify the alg attribute in the token header, then delete HS256, set it to none, and use an empty signature (e.g., signature = ""). Use this token and replay it in a request. Some libraries treat tokens signed with the none algorithm as a valid token with a verified signature. This allows attackers to create their own "signed" tokens.

There are two different Burp Plugins that can help you for testing the vulnerabilities listed above:

- JSON Web Token Attacker
- JSON Web Tokens

Also, make sure to check out the OWASP JWT Cheat Sheet for additional information.

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Dynamic Analysis

4.3.4 Testing OAuth 2.0 Flows

OAuth 2.0 defines a delegation protocol for conveying authorization decisions across APIs and a network of web-enabled applications. It is used in a variety of applications, including user authentication applications.

Common uses for OAuth2 include:

- Getting permission from the user to access an online service using their account.
- Authenticating to an online service on behalf of the user.
- Handling authentication errors.

According to OAuth 2.0, a mobile client seeking access to a user's resources must first ask the user to authenticate against an authentication server. With the users' approval, the authorization server then issues a token that allows the app to act on behalf of the user. Note that the OAuth2 specification doesn't define any particular kind of authentication or access token format.

OAuth 2.0 defines four roles:

- Resource Owner: the account owner
- Client: the application that wants to access the user's account with the access tokens
- Resource Server: hosts the user accounts
- Authorization Server: verifies user identity and issues access tokens to the application

Note: The API fulfills both the Resource Owner and Authorization Server roles. Therefore, we will refer to both as the API.

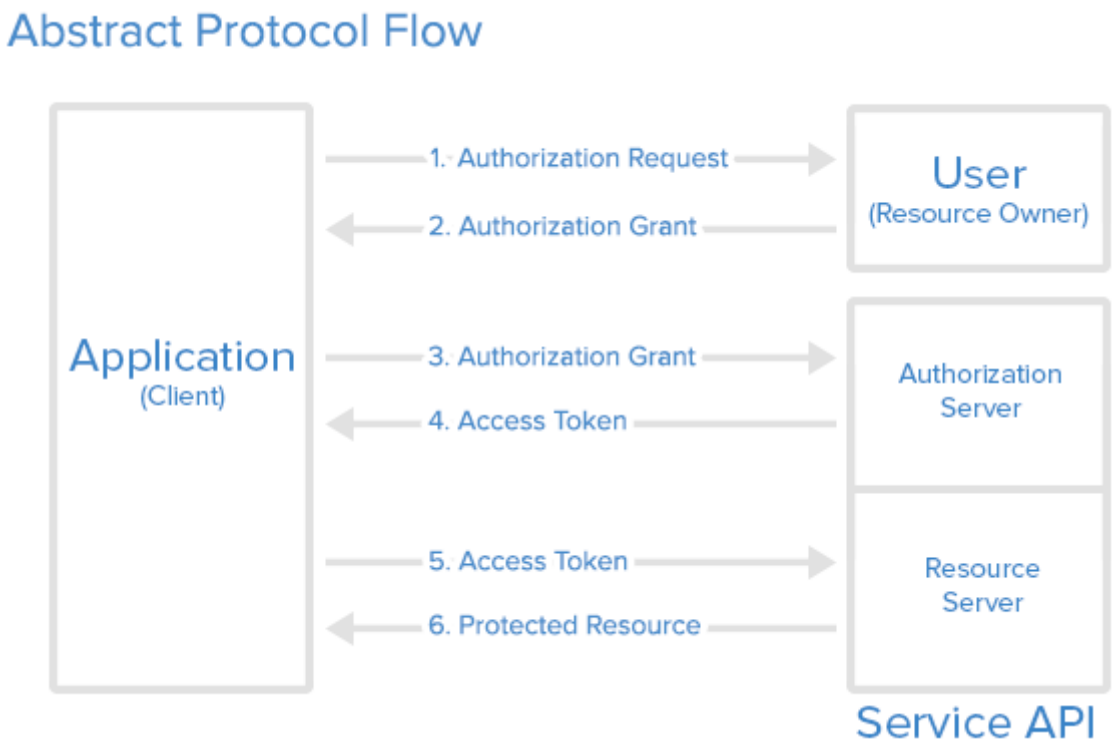


Fig 4.3.4.1 Abstract Protocol Flow

Here is a more detailed explanation of the steps in the diagram:

1. The application requests user authorization to access service resources.
2. If the user authorizes the request, the application receives an authorization grant. The authorization grant may take several forms (explicit, implicit, etc.).

3. The application requests an access token from the authorization server (API) by presenting authentication of its own identity along with the authorization grant.
4. If the application identity is authenticated and the authorization grant is valid, the authorization server (API) issues an access token to the application, completing the authorization process. The access token may have a companion refresh token.
5. The application requests the resource from the resource server (API) and presents the access token for authentication. The access token may be used in several ways (e.g., as a bearer token).
6. If the access token is valid, the resource server (API) serves the resource to the application.

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-1 and MSTG-AUTH-3)

4.3.5 OAuth 2.0 Best Practices

Verify that the following best practices are followed:

User agent:

- The user should have a way to visually verify trust (e.g., Transport Layer Security (TLS) confirmation, website mechanisms).
- To prevent man-in-the-middle attacks, the client should validate the server's fully qualified domain name with the public key the server presented when the connection was established.

Type of grant:

- On native apps, code grant should be used instead of implicit grant.
- When using code grant, PKCE (Proof Key for Code Exchange) should be implemented to protect the code grant. Make sure that the server also implements it.
- The auth "code" should be short-lived and used immediately after it is received. Verify that auth codes only reside on transient memory and aren't stored or logged.

Client secrets:

- Shared secrets should not be used to prove the client's identity because the client could be impersonated ("client_id" already serves as proof). If they do use client secrets, be sure that they are stored in secure local storage.

End-User credentials:

- Secure the transmission of end-user credentials with a transport-layer method, such as TLS.

Tokens:

- Keep access tokens in transient memory.
- Access tokens must be transmitted over an encrypted connection.
- Reduce the scope and duration of access tokens when end-to-end confidentiality can't be guaranteed or the token provides access to sensitive information or transactions.
- Remember that an attacker who has stolen tokens can access their scope and all resources associated with them if the app uses access tokens as bearer tokens with no other way to identify the client.
- Store refresh tokens in secure local storage; they are long-term credentials.

Reference

- owasp-mastg Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3) OAUTH 2.0 Best Practices

Rulebook

- *Adherence to OAuth 2.0 best practices (Required)*

4.3.5.1 External User Agent vs. Embedded User Agent

OAuth2 authentication can be performed either through an external user agent (e.g. Chrome or Safari) or in the app itself (e.g. through a WebView embedded into the app or an authentication library). None of the two modes is intrinsically "better" - instead, what mode to choose depends on the context.

Using an external user agent is the method of choice for apps that need to interact with social media accounts (Facebook, Twitter, etc.). Advantages of this method include:

- The user's credentials are never directly exposed to the app. This guarantees that the app cannot obtain the credentials during the login process ("credential phishing").
- Almost no authentication logic must be added to the app itself, preventing coding errors.

On the negative side, there is no way to control the behavior of the browser (e.g. to activate certificate pinning).

For apps that operate within a closed ecosystem, embedded authentication is the better choice. For example, consider a banking app that uses OAuth2 to retrieve an access token from the bank's authentication server, which is then used to access a number of micro services. In that case, credential phishing is not a viable scenario. It is likely preferable to keep the authentication process in the (hopefully) carefully secured banking app, instead of placing trust on external components.

Reference

- owasp-mastg Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3) External User Agent vs. Embedded User Agent

Rulebook

- *Main recommended libraries used for OAuth2 authentication (Recommended)*
- *When using an external user agent for OAuth2 authentication, understand the advantages and disadvantages and use it (Recommended)*

4.3.6 Other OAuth 2.0 Best Practices

For additional best practices and detailed information please refer to the following source documents:

- RFC6749 - The OAuth 2.0 Authorization Framework (October 2012)
- RFC8252 - OAuth 2.0 for Native Apps (October 2017)
- RFC6819 - OAuth 2.0 Threat Model and Security Considerations (January 2013)

Reference

- owasp-mastg Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3) Other OAuth2 Best Practices

4.3.7 Rulebook

1. *Identify JWT libraries in use and whether they have known vulnerabilities (Required)*
2. *Ensure tokens are securely stored on mobile devices by KeyStore (Android) (Required)*
3. *Need to request the expected hash algorithm (Required)*
4. *Set an appropriate period for refresh token expiration (Required)*
5. *Adherence to OAuth 2.0 best practices (Required)*
6. *Main recommended libraries used for OAuth2 authentication (Recommended)*
7. *When using an external user agent for OAuth2 authentication, understand the advantages and disadvantages and use it (Recommended)*

4.3.7.1 Identify JWT libraries in use and whether they have known vulnerabilities (Required)

The following information should be reviewed to determine if there are any vulnerabilities.

- Identify the JWT libraries you are using and find out whether the JWT libraries in use have any known vulnerabilities.
- Ensure that the HMAC is verified for all incoming requests containing tokens.
- Verify the location of the private signing key or HMAC private key. Keys should be stored on the server and never shared with clients. It should only be available to the issuer and verifier.
- Ensure that no sensitive data, such as personally identifiable information, is embedded in the JWT. For architectures that need to transmit such information in the token for any reason, ensure that payload encryption is applied. See the sample Java implementation in OWASP JWT Cheat Sheet.
- Ensure that replay attacks are addressed with jti (JWT ID) claims that give the JWT a unique identifier.
- Ensure that cross-service relay attacks are addressed with aud (audience) claims that define which application the token is for.
- Ensure that the token is securely stored on the mobile device by means of KeyStore (Android).

The sample code below is an example of storing a token encrypted with the KeyStore encryption key in SharedPreferences and retrieving it from SharedPreferences for decryption.

```
val PROVIDER = "AndroidKeyStore"
val ALGORITHM = "RSA"
val CIPHER_TRANSFORMATION = "RSA/ECB/PKCS1Padding"

/**
 * Encrypt tokens and save them to SharedPreferences
 */
fun saveToken(context: Context, token: String) {
    val encryptedToken = encrypt(context, "token", token)
    val editor = getSharedPreferences(context).edit()
    editor.putString("encrypted_token", encryptedToken).commit()
}

/**
 * Decrypt the token and get it from SharedPreferences
 */
fun getToken(context: Context): String? {
    val encryptedToken = getSharedPreferences(context).getString("encrypted_token",
↪ null)
    if (encryptedToken == null) {
        return encryptedToken
    }
    val token = decrypt("token", encryptedToken)
    return token
}

/**
 * Encrypt text
 * @param context
 * @param alias Elias for identifying symmetric keys. Unique for each use.
 * @param plainText Text to be encrypted
 * @return Encrypted and Base64 wrapped string
 */
fun encrypt(context: Context, alias: String, plainText: String): String {
    val keyStore = KeyStore.getInstance(PROVIDER)
    keyStore.load(null)

    // Generate if no symmetric key
    if (!keyStore.containsAlias(alias)) {
```

(continues on next page)

(continued from previous page)

```
        val keyPairGenerator = KeyPairGenerator.getInstance(ALGORITHM, PROVIDER)
        keyPairGenerator.initialize(createKeyPairGeneratorSpec(context, alias))
        keyPairGenerator.generateKeyPair()
    }
    val publicKey = keyStore.getCertificate(alias).getPublicKey()
    val privateKey = keyStore.getKey(alias, null)

    // Encryption with public key
    val cipher = Cipher.getInstance(CIPHER_TRANSFORMATION)
    cipher.init(Cipher.ENCRYPT_MODE, publicKey)
    val bytes = cipher.doFinal(plainText.toByteArray(Charsets.UTF_8))

    // String in Base64 for easy storage in SharedPreferences
    return Base64.encodeToString(bytes, Base64.DEFAULT)
}

/**
 * decrypt encrypted text.
 * @param alias alias to identify the symmetric key. Make it unique for each use.
 * @param encryptedText Text encrypted with encrypt
 * @return decrypted text
 */
fun decrypt(alias: String, encryptedText: String): String? {
    val keyStore = KeyStore.getInstance(PROVIDER)
    keyStore.load(null)
    if (!keyStore.containsAlias(alias)) {
        return null
    }

    // Decrypt with private key
    val privateKey = keyStore.getKey(alias, null)
    val cipher = Cipher.getInstance(CIPHER_TRANSFORMATION)
    cipher.init(Cipher.DECRYPT_MODE, privateKey)
    val bytes = Base64.decode(encryptedText, Base64.DEFAULT)

    val b = cipher.doFinal(bytes)
    return String(b)
}

/**
 * generate a symmetric key
 */
fun createKeyPairGeneratorSpec(context: Context, alias: String):␣
↪KeyPairGeneratorSpec {
    val start = Calendar.getInstance()
    val end = Calendar.getInstance()
    end.add(Calendar.YEAR, 100)

    return KeyPairGeneratorSpec.Builder(context)
            .setAlias(alias)
            .setSubject(X500Principal(String.format("CN=%s", alias)))
            .setSerialNumber(BigInteger.valueOf(1000000))
            .setStartDate(start.getTime())
            .setEndDate(end.getTime())
            .build()
}
```

If this is violated, the following may occur.

- A vulnerability related to the JWT library used is exploited.

4.3.7.2 Ensure tokens are securely stored on mobile devices by KeyStore (Android) (Required)

Tokens must be securely stored on the mobile device by KeyStore. Failure to do so may allow server authentication using tokens by a third party.

Refer to the following rules for how to use KeyStore.

Data Storage and_Privacy Requirements RuleBook

- *Verify that keys are stored inside security hardware (Recommended)*

If this is violated, the following may occur.

- Server authentication using a token by a third party becomes possible.

4.3.7.3 Need to request the expected hash algorithm (Required)

An attacker may modify the token and change the signature algorithm using the "none" keyword to indicate that the integrity of the token has already been verified. Therefore, Java applications should explicitly request the expected algorithm when creating the verification context.

```
// HMAC key - Block serialization and storage as String in JVM memory
private transient byte[] keyHMAC = ...;

//Create a verification context for the token requesting explicitly the use of the␣
↪HMAC-256 HMAC generation

JWTVerifier verifier = JWT.require(Algorithm.HMAC256(keyHMAC)).build();

//Verify the token; if the verification fails then an exception is thrown

DecodedJWT decodedToken = verifier.verify(token);
```

If this is violated, the following may occur.

- Treat tokens signed with the " none " algorithm as if they were valid tokens with verified signatures, and applications may trust requests for modified tokens.

4.3.7.4 Set an appropriate period for refresh token expiration (Required)

It is necessary to ensure that the refresh tokens are valid for the appropriate period of time.If the refresh token is within the expiration date, a new access token should be issued; if it is outside the expiration date, there is a need to deny access and prompt the user to log in again.

If this is violated, the following may occur.

- If the expiration date is inappropriate, there is a possibility that the login user can be used by a third party.

4.3.7.5 Adherence to OAuth 2.0 best practices (Required)

Adhere to the following OAuth 2.0 best practices

User agent

- Users should have a way to visually confirm trust (e.g. Transport Layer Security (TLS) confirmation, website mechanisms).
- To prevent man-in-the-middle attacks, the client must verify the fully qualified domain name of the server with the public key presented by the server when establishing the connection.

Type of grant

- Native apps should use code assignment rather than implicit assignment.
- If code assignment is used, PKCE (Proof Key for Code Exchange) must be implemented to protect the code assignment. Make sure it is also implemented on the server side.

- Authentication "codes" should have a short expiration date and be used immediately upon receipt. Ensure that the authentication code exists only in temporary memory and is not stored or logged.

Client secret

- The client should not use the shared secret to prove the client's identity, since it can be spoofed ("client_id" already serves as proof). If the client secret is used, make sure it is stored in secure local storage.

End User Authentication Information

- Transport layer methods such as TLS protect the transmission of end-user authentication information.

token

- The access token is stored in temporary memory.
- The access token must be sent over an encrypted connection.
- Reduce the scope and duration of the access token if end-to-end confidentiality cannot be guaranteed or if the token provides access to sensitive information or transactions.
- Note that if the application uses the access token as a bearer token and there is no other way to identify the client, an attacker who steals the token will have access to its scope and all resources associated with it.
- Refresh tokens should be stored in secure local storage.

If this is violated, the following may occur.

- Potentially a vulnerable OAuth 2.0 implementation.

4.3.7.6 Main recommended libraries used for OAuth2 authentication (Recommended)

It is important not to develop your own OAuth2 authentication implementation.
The main libraries used to implement OAuth2 authentication are

- Google Play Services (Google)

The sample code below is an example of OAuth2 authentication with Google Play Services.

```
/**
 * Google SignIn
 */
private fun requestOAuth() {
    val signInOptions = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_
↪SIGN_IN).build()
    activityResultLauncher.launch(GoogleSignIn.getClient(requireContext(),
↪signInOptions).signInIntent)
}
```

If this is not noted, the following may occur.

- Potentially a vulnerable OAuth 2.0 implementation.

4.3.7.7 When using an external user agent for OAuth2 authentication, understand the advantages and disadvantages and use it (Recommended)

OAuth2 authentication can be performed via an external user agent (e.g. Chrome or Safari). The advantages and disadvantages of using an external user agent are as follows

Advantages

- The user's credentials are never disclosed directly to the app. For this reason, it is not possible for an app to obtain authentication information when logging in ("authentication information phishing").
- Since little or no authentication logic needs to be added to the application itself, coding errors can be avoided.

Disadvantage

- There is no way to control browser behavior (e.g., enable certificate pinning).

If this is not noted, the following may occur.

- Not available when certificate pinning needs to be enabled.

4.4 MSTG-AUTH-4

The remote endpoint terminates the existing session when the user logs out.

4.4.1 Destroying Remote Session Information

The purpose of this test case is verifying logout functionality and determining whether it effectively terminates the session on both client and server and invalidates a stateless token.

Failing to destroy the server-side session is one of the most common logout functionality implementation errors. This error keeps the session or token alive, even after the user logs out of the application. An attacker who gets valid authentication information can continue to use it and hijack a user's account.

Many mobile apps don't automatically log users out. There can be various reasons, such as: because it is inconvenient for customers, or because of decisions made when implementing stateless authentication. The application should still have a logout function, and it should be implemented according to best practices, destroying all locally stored tokens or session identifiers. If session information is stored on the server, it should also be destroyed by sending a logout request to that server. In case of a high-risk application, tokens should be invalidated. Not removing tokens or session identifiers can result in unauthorized access to the application in case the tokens are leaked. Note that other sensitive types of information should be removed as well, as any information that is not properly cleared may be leaked later, for example during a device backup.

4.4.2 Static Analysis

If server code is available, make sure logout functionality terminates the session correctly. This verification will depend on the technology. Here are different examples of session termination for proper server-side logout:

- Spring (Java)
- Ruby on Rails
- PHP

If access and refresh tokens are used with stateless authentication, they should be deleted from the mobile device. The refresh token should be invalidated on the server.

4.4.3 Dynamic Analysis

Use an interception proxy for dynamic application analysis and execute the following steps to check whether the logout is implemented properly:

1. Log in to the application.
2. Access a resource that requires authentication, typically a request for private information belonging to your account.
3. Log out of the application.
4. Try to access the data again by resending the request from step 2.

If the logout is correctly implemented on the server, an error message or redirect to the login page will be sent back to the client. On the other hand, if you receive the same response you got in step 2, the token or session ID is still valid and hasn't been correctly terminated on the server. The OWASP Web Testing Guide (WSTG-SESS-06) includes a detailed explanation and more test cases.

Reference

- owasp-mastg Testing User Logout (MSTG-AUTH-4)

Rulebook

- *Best practices for discarding remote session information when login functionality is present in the app (Recommended)*

4.4.4 Rulebook

1. *Best practices for discarding remote session information when login functionality is present in the app (Recommended)*

4.4.4.1 Best practices for discarding remote session information when login functionality is present in the app (Recommended)

If login capability exists in the app, destroy remote session information according to the following best practices.

- Provide a logout function in the application.
- Discard all locally stored tokens and session identifiers upon logout.
- If session information is stored on the server, send a logout request to the server and destroy it.
- Disable tokens for high-risk applications.

Failure to destroy tokens and session identifiers could result in unauthorized access to the application if the tokens are leaked. Information that is not destroyed should be destroyed in the same manner as other sensitive information, as it could be leaked later, such as when backing up a device.

If this is not noted, the following may occur.

- Login information remains in memory and can be used by third parties.

4.5 MSTG-AUTH-5

A password policy exists and is enforced at the remote endpoint.

4.5.1 Password Policy Compliance

Password strength is a key concern when passwords are used for authentication. The password policy defines requirements to which end users should adhere. A password policy typically specifies password length, password complexity, and password topologies. A "strong" password policy makes manual or automated password cracking difficult or impossible. The following sections will cover various areas regarding password best practices. For further information please consult the OWASP Authentication Cheat Sheet.

Reference

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6)

4.5.2 Static Analysis

Confirm the existence of a password policy and verify the implemented password complexity requirements according to the OWASP Authentication Cheat Sheet which focuses on length and an unlimited character set. Identify all password-related functions in the source code and make sure that a verification check is performed in each of them. Review the password verification function and make sure that it rejects passwords that violate the password policy.

Reference

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) Static Analysis

Rulebook

- *A "strong" password policy (Recommended)*

4.5.2.1 zxcvbn

zxcvbn is a common library that can be used for estimating password strength, inspired by password crackers. It is available in JavaScript but also for many other programming languages on the server side. There are different methods of installation, please check the Github repo for your preferred method. Once installed, zxcvbn can be used to calculate the complexity and the amount of guesses to crack the password.

After adding the zxcvbn JavaScript library to the HTML page, you can execute the command zxcvbn in the browser console, to get back detailed information about how likely it is to crack the password including a score.

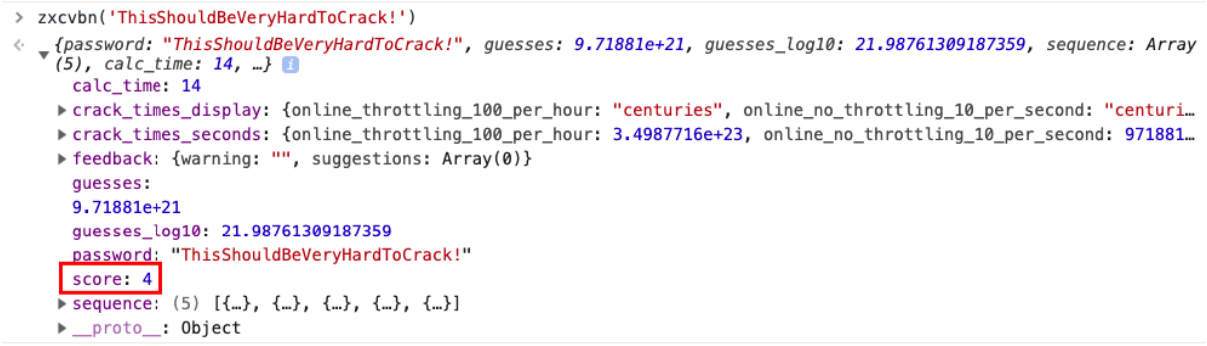


Fig 4.5.2.1.1 zxcvbn Command Example

The score is defined as follows and can be used for a password strength bar for example:

```
0 # too guessable: risky password. (guesses < 10^3)

1 # very guessable: protection from throttled online attacks. (guesses < 10^6)

2 # somewhat guessable: protection from unthrottled online attacks. (guesses < 10^
↪8)

3 # safely unguessable: moderate protection from offline slow-hash scenario.␣
↪(guesses < 10^10)

4 # very unguessable: strong protection from offline slow-hash scenario. (guesses >
↪= 10^10)
```

Note that zxcvbn can be implemented by the app-developer as well using the Java (or other) implementation in order to guide the user into creating a strong password.

Reference

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) Static Analysis zxcvbn

4.5.3 Have I Been Pwned: PwnedPasswords

In order to further reduce the likelihood of a successful dictionary attack against a single factor authentication scheme (e.g. password only), you can verify whether a password has been compromised in a data breach. This can be done using services based on the Pwned Passwords API by Troy Hunt (available at api.pwnedpasswords.com). For example, the "Have I been pwned?" companion website. Based on the SHA-1 hash of a possible password candidate, the API returns the number of times the hash of the given password has been found in the various breaches collected by the service. The workflow takes the following steps:

- Encode the user input to UTF-8 (e.g.: the password test).
- Take the SHA-1 hash of the result of step 1 (e.g.: the hash of test is A94A8FE5CC⋯).
- Copy the first 5 characters (the hash prefix) and use them for a range-search by using the following API: http GET https://api.pwnedpasswords.com/range/A94A8
- Iterate through the result and look for the rest of the hash (e.g. is FE5CC⋯ part of the returned list?). If it is not part of the returned list, then the password for the given hash has not been found. Otherwise, as in case of FE5CC⋯, it will return a counter showing how many times it has been found in breaches (e.g.: FE5CC⋯:76479).

Further documentation on the Pwned Passwords API can be found online.

Note that this API is best used by the app-developer when the user needs to register and enter a password to check whether it is a recommended password or not.

Reference

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) Have I Been Pwned: PwnedPasswords

Rulebook

- *Check if the password is recommended (Recommended)*

4.5.3.1 Login limit

Check the source code for a throttling procedure: a counter for logins attempted in a short period of time with a given user name and a method to prevent login attempts after the maximum number of attempts has been reached. After an authorized login attempt, the error counter should be reset.

Observe the following best practices when implementing anti-brute-force controls:

- After a few unsuccessful login attempts, targeted accounts should be locked (temporarily or permanently), and additional login attempts should be rejected.
- A five-minute account lock is commonly used for temporary account locking.
- The controls must be implemented on the server because client-side controls are easily bypassed.
- Unauthorized login attempts must be tallied with respect to the targeted account, not a particular session.

Additional brute force mitigation techniques are described on the OWASP page Blocking Brute Force Attacks.

Reference

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) Have I Been Pwned: PwnedPasswords Login Throttling

Rulebook

- *Adhere to the following best practices for brute force protection (Required)*

4.5.4 Rulebook

1. *A "strong" password policy (Recommended)*
2. *Check if the password is recommended (Recommended)*
3. *Adhere to the following best practices for brute force protection (Required)*

4.5.4.1 A "strong" password policy (Recommended)

The following is an example of a policy for setting a "strong" password.

- Password Length
  - Minimum: Passwords of less than 8 characters are considered weak.
  - Maximum: The typical maximum length is 64 characters. It is necessary to set the maximum length to prevent long password Denial of Service attacks.
- Do not truncate passwords without notifying the user.
- Password configuration rulesAllow the use of all characters, including Unicode and whitespace.
- Credential RotationCredential rotation should be performed when a password compromise occurs or is identified.
- Password Strength MeterPrepare a password strength meter to block users from creating complex passwords and setting passwords that have been identified in the past.

The sample code below is an example of the password length determination and password composition rule process that should be supported by an Android application.

Example in layout:

```
<EditText
    android:id="@+id/passwordText"
    android:hint="password"
    android:maxLength="64"/>
```

Example in Kotlin:

```
val PASSWORD_MIN_LENGTH: Int = 8
val PASSWORD_MAX_LENGTH: Int = 64
fun validationPassword(password: String?): Boolean {
    return if (password == null || password.isEmpty()) {
        false
    } else !(password.length < PASSWORD_MIN_LENGTH || password.length > PASSWORD_
↪MAX_LENGTH)
}
```

Reference

- OWASP CheatSheetSeries Implement Proper Password Strength Controls

If this is not noted, the following may occur.

- Predicted to be a weak password.

4.5.4.2 Check if the password is recommended (Recommended)

The Pwned Passwords API can be used to check if a password is recommended when a user registers or enters a password.

If this is not noted, the following may occur.

- Predicted to be a weak password.

4.5.4.3 Adhere to the following best practices for brute force protection (Required)

The following best practices should be followed as a brute force measure.

- After several failed login attempts, the target account should be locked (temporarily or permanently) and subsequent login attempts should be denied.
- A 5-minute account lock is commonly used for temporary account locks.
- Control should be on the server, as client-side controls are easily bypassed.
- Unauthorized login attempts should be aggregated for the target account, not for a specific session.

Additional brute force mitigation techniques are described on the OWASP page Blocking Brute Force Attacks.

* Sample code is not available, as these are server-side rules.

If this is violated, the following may occur.

- Become vulnerable to brute force attacks.

4.6 MSTG-AUTH-6

The remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times.

* This guide is omitted in this document because it is a chapter about support on the Remote Service side.

Reference

- owasp-mastg Testing Best Practices for Passwords Dynamic Testing(MSTG-AUTH-6)

4.7 MSTG-AUTH-7

Sessions are invalidated at the remote endpoint after a predefined period of inactivity and access tokens expire.

* This guide is omitted in this document because it is a chapter about support on the Remote Service side.

Reference

- owasp-mastg Testing Session Timeout(MSTG-AUTH-7)

4.8 MSTG-AUTH-12

Authorization models should be defined and enforced at the remote endpoint.

* This guide is omitted in this document because it is a chapter about support on the Remote Service side.

5 Network Communication Requirements

5.1 MSTG-NETWORK-1

Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.

5.1.1 Secure Network Requests

5.1.1.1 Recommended Network APIs

First, you should identify all network requests in the source code and ensure that no plain HTTP URLs are used. Make sure that sensitive information is sent over secure channels by using HttpsURLConnection or SSLSocket (for socket-level communication using TLS).

Next, even when using a low-level API which is supposed to make secure connections (such as SSLSocket), be aware that it has to be securely implemented. For instance, SSLSocket doesn't verify the hostname. Use getDefaultHostnameVerifier to verify the hostname. The Android developer documentation includes a code example.

Reference

- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1) Testing Network Requests over Secure Protocols
- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1) Testing Network API Usage

Rulebook

- *Ensure that plain-text HTTP URLs are not used (Required)*
- *Ensure that sensitive information is transmitted over a secure channel (Required)*
- *Secure implementation using low-level API (Required)*

5.1.1.2 Configure plain-text HTTP Traffic

Next, you should ensure that the app is not allowing cleartext HTTP traffic. Since Android 9 (API level 28) cleartext HTTP traffic is blocked by default (thanks to the default Network Security Configuration) but there are multiple ways in which an application can still send it:

- Setting the android:usesCleartextTraffic attribute of the <application> tag in the AndroidManifest.xml file. Note that this flag is ignored in case the Network Security Configuration is configured.
- Configuring the Network Security Configuration to enable cleartext traffic by setting the cleartextTrafficPermitted attribute to true on <domain-config> elements.
- Using low-level APIs (e.g. Socket) to set up a custom HTTP connection.

- Using a cross-platform framework (e.g. Flutter, Xamarin, ⋯), as these typically have their own implementations for HTTP libraries.

All of the above cases must be carefully analyzed as a whole. For example, even if the app does not permit cleartext traffic in its Android Manifest or Network Security Configuration, it might actually still be sending HTTP traffic. That could be the case if it's using a low-level API (for which Network Security Configuration is ignored) or a badly configured cross-platform framework.

For more information refer to the article "Security with HTTPS and SSL".

Reference

- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1) Testing for Cleartext Traffic

Rulebook

- *Ensure that the app does not allow plain-text HTTP traffic (Required)*

5.1.2 Rulebook

1. *Ensure that plain-text HTTP URLs are not used (Required)*
2. *Ensure that sensitive information is transmitted over a secure channel (Required)*
3. *Secure implementation using low-level API (Required)*
4. *Ensure that the app does not allow plain-text HTTP traffic (Required)*

5.1.2.1 Ensure that plain-text HTTP URLs are not used (Required)

It is necessary to identify all network requests in the source code and ensure that no plain-text HTTP URLs are used.

If this is violated, the following may occur.

- Leakage of plain-text information to third parties.

5.1.2.2 Ensure that sensitive information is transmitted over a secure channel (Required)

If confidential information is sent through a dangerous channel (HTTP), it may be leaked to a third party because it is sent in plain text. Therefore, when sending confidential information, it must be sent over a secure channel (HTTPS, SSL, etc.).

The following is a sample code for transmission over a secure channel.

- HttpsURLConnection

```
val url = URL("https://gmail.com:433/")
val urlConnection = url.openConnection() as HttpsURLConnection
urlConnection.connect();
```

- SSLSocket

```
val socket: SSLSocket = SSLSocketFactory.getDefault().run {
    createSocket("gmail.com", 443) as SSLSocket
}
```

If this is violated, the following may occur.

- Confidential information is leaked to a third party.

5.1.2.3 Secure implementation using low-level API (Required)

Even when using low-level APIs, secure implementations are required. SSLSocket does not validate hostnames. To verify the host name, use getDefaultHostnameVerifier.

The following is an example of sample code for host name verification when using SSLSocket.

```
    // Open SSLSocket directly to gmail.com
    val socket: SSLSocket = SSLSocketFactory.getDefault().run {
        createSocket("gmail.com", 443) as SSLSocket
    }
    val session = socket.session

    // Verify that the certicate hostname is for mail.google.com
    // This is due to lack of SNI support in the current SSLSocket.
    HttpsURLConnection.getDefaultHostnameVerifier().run {
        if (!verify("mail.google.com", session)) {
            throw SSLHandshakeException("Expected mail.google.com, found ${session.
↪peerPrincipal} ")
        }
    }

    // At this point SSLSocket performed certificate verification and
    // we have performed hostname verification, so it is safe to proceed.

    // ... use socket ...
    socket.close()
```

If this is violated, the following may occur.

- The host to which you are communicating may not be trusted or guaranteed.

5.1.2.4 Ensure that the app does not allow plain-text HTTP traffic (Required)

Ensure that the app does not allow plaintext HTTP traffic. Since Android 9 (API level 28), plain-text HTTP traffic is blocked by default, but there are multiple ways for apps to send plain text.

The following is an example of how an app can send plain text.

- In the AndroidManifest.xml file, the <application> tag Set the android:usesCleartextTraffic attribute in the AndroidManifest.xml file. Note that this flag is ignored if Network Security Configuration is set.

```
<application
            android:usesCleartextTraffic="true">
</application>
```

- Set Network Security Configuration to enable CleartextTraffic by setting the cleartextTrafficPermitted attribute to true in the <domain-config> element.

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <base-config cleartextTrafficPermitted="false" />
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">secure.example.com</domain>
    </domain-config>
</network-security-config>
```

- Set up a custom HTTP connection using a low-level API (e.g., Socket).

```
val address = InetSocketAddress(ip, port)
val socket = Socket()
try {
```

(continues on next page)

(continued from previous page)

```
    socket.connect(address)
} catch (e: Exception) {
}
```

- Use a cross-platform framework (Flutter, Xamarin, etc.). These usually have their own implementations of HTTP libraries.

If this is violated, the following may occur.

- Send plaintext over HTTP traffic.

5.2 MSTG-NETWORK-2

The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards.

5.2.1 Recommended TLS Settings

Ensuring proper TLS configuration on the server side is also important. The SSL protocol is deprecated and should no longer be used. Also TLS v1.0 and TLS v1.1 have known vulnerabilities and their usage is deprecated in all major browsers by 2020. TLS v1.2 and TLS v1.3 are considered best practice for secure transmission of data. Starting with Android 10 (API level 29) TLS v1.3 will be enabled by default for faster and secure communication. The major change with TLS v1.3 is that customizing cipher suites is no longer possible and that all of them are enabled when TLS v1.3 is enabled, whereas Zero Round Trip (0-RTT) mode isn't supported.

When both the client and server are controlled by the same organization and used only for communicating with one another, you can increase security by hardening the configuration.

If a mobile application connects to a specific server, its networking stack can be tuned to ensure the highest possible security level for the server's configuration. Lack of support in the underlying operating system may force the mobile application to use a weaker configuration.

Reference

- owasp-mastg Verifying the TLS Settings (MSTG-NETWORK-2) Recommended TLS Settings

Rulebook

- *Secure communication protocol (Required)*

5.2.2 Recommended Cipher Suites

Cipher suites have the following structure:

```
Protocol_KeyExchangeAlgorithm_WITH_BlockCipher_IntegrityCheckAlgorithm
```

This structure includes:

- A Protocol used by the cipher
- A Key Exchange Algorithm used by the server and the client to authenticate during the TLS handshake
- A Block Cipher used to encrypt the message stream
- A Integrity Check Algorithm used to authenticate messages

Example: TLS_RSA_WITH_3DES_EDE_CBC_SHA

In the example above the cipher suites uses:

- TLS as protocol

- RSA Asymmetric encryption for Authentication
- 3DES for Symmetric encryption with EDE_CBC mode
- SHA Hash algorithm for integrity

Note that in TLSv1.3 the Key Exchange Algorithm is not part of the cipher suite, instead it is determined during the TLS handshake.

In the following listing, we' ll present the different algorithms of each part of the cipher suite.

Protocols:

- SSLv1
- SSLv2 - RFC 6176
- SSLv3 - RFC 6101
- TLSv1.0 - RFC 2246
- TLSv1.1 - RFC 4346
- TLSv1.2 - RFC 5246
- TLSv1.3 - RFC 8446

Key Exchange Algorithms:

- DSA - RFC 6979
- ECDSA - RFC 6979
- RSA - RFC 8017
- DHE - RFC 2631 - RFC 7919
- ECDHE - RFC 4492
- PSK - RFC 4279
- DSS - FIPS186-4
- DH_anon - RFC 2631 - RFC 7919
- DHE_RSA - RFC 2631 - RFC 7919
- DHE_DSS - RFC 2631 - RFC 7919
- ECDHE_ECDSA - RFC 8422
- ECDHE_PSK - RFC 8422 - RFC 5489
- ECDHE_RSA - RFC 8422

Block Ciphers:

- DES - RFC 4772
- DES_CBC - RFC 1829
- 3DES - RFC 2420
- 3DES_EDE_CBC - RFC 2420
- AES_128_CBC - RFC 3268
- AES_128_GCM - RFC 5288
- AES_256_CBC - RFC 3268
- AES_256_GCM - RFC 5288
- RC4_40 - RFC 7465
- RC4_128 - RFC 7465

- CHACHA20_POLY1305 - RFC 7905 - RFC 7539

Integrity Check Algorithms:

- MD5 - RFC 6151
- SHA - RFC 6234
- SHA256 - RFC 6234
- SHA384 - RFC 6234

Note that the efficiency of a cipher suite depends on the efficiency of its algorithms.

The following resources contain the latest recommended cipher suites to use with TLS:

- IANA recommended cipher suites can be found in TLS Cipher Suites.
- OWASP recommended cipher suites can be found in the TLS Cipher String Cheat Sheet.

Some Android versions do not support some of the recommended cipher suites, so for compatibility purposes you can check the supported cipher suites for Android versions and choose the top supported cipher suites.

If you want to verify whether your server supports the right cipher suites, there are various tools you can use:

- testssl.sh which "is a free command line tool which checks a server' s service on any port for the support of TLS/SSL ciphers, protocols as well as some cryptographic flaws" .

Finally, verify that the server or termination proxy at which the HTTPS connection terminates is configured according to best practices. See also the OWASP Transport Layer Protection cheat sheet and the Qualys SSL/TLS Deployment Best Practices.

Reference

- owasp-mastg Verifying the TLS Settings (MSTG-NETWORK-2) Cipher Suites Terminology

Rulebook

- *Recommended cipher suites for TLS (Recommended)*

5.2.3 Rulebook

1. *Secure communication protocol (Required)*
2. *Recommended cipher suites for TLS (Recommended)*

5.2.3.1 Secure communication protocol (Required)

Ensuring proper TLS configuration on the server side is also important. The SSL protocol is deprecated and should no longer be used.

Deprecated Protocols

- SSL
- TLS v1.0
- TLS v1.1

TLS v1.0 and TLS v1.1 have been deprecated in all major browsers by 2020.

Recommended Protocols

- TLS v1.2
- TLS v1.3

Starting with Android 10 (API level 29), TLS v1.3 is enabled by default for faster and more secure communication. While enabling TLS v1.3 enables all cipher suites, 0-RTT (Zero Round Trip) mode is not supported.

If this is violated, the following may occur.

- Vulnerable to security exploits.

5.2.3.2 Recommended cipher suites for TLS (Recommended)

The following is an example of a recommended cipher suite. (Cipher suites recommended by TLS Cipher Suites that are not deprecated by Android that are not deprecated).

- TLS_DHE_PSK_WITH_AES_128_GCM_SHA256
- TLS_DHE_PSK_WITH_AES_256_GCM_SHA384
- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256
- TLS_AES_128_CCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_DHE_RSA_WITH_AES_128_CCM
- TLS_DHE_RSA_WITH_AES_256_CCM
- TLS_DHE_PSK_WITH_AES_128_CCM
- TLS_DHE_PSK_WITH_AES_256_CCM
- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_PSK_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_PSK_WITH_AES_128_CCM_SHA256

If this is not noted, the following may occur.

- Potential use of weak cipher suites.

5.3 MSTG-NETWORK-3

The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted.

5.3.1 Configuring Trusted Certificates

5.3.1.1 Default settings per target SDK version

Applications targeting Android 7.0 (API level 24) or higher will use a default Network Security Configuration that doesn't trust any user supplied CAs, reducing the possibility of MITM attacks by luring users to install malicious CAs.

Decode the app using apktool and verify that the targetSdkVersion in apktool.yml is equal to or higher than 24.

```
grep targetSdkVersion UnCrackable-Level3/apktool.yml
  targetSdkVersion: '28'
```

However, even if targetSdkVersion >=24, the developer can disable default protections by using a custom Network Security Configuration defining a custom trust anchor forcing the app to trust user supplied CAs. See "*Analyzing Custom Trust Anchors*".

Reference

- owasp-mastg Testing Endpoint Identify Verification (MSTG-NETWORK-3) Static Analysis Verifying the Target SDK Version

Rulebook

- *MITM attack potential depending on target SDK version (Required)*
- *Custom trust anchor analysis (Required)*

5.3.1.2 Analyzing Custom Trust Anchors

Search for the Network Security Configuration file and inspect any custom <trust-anchors> defining <certificates src="user"> (which should be avoided).

You should carefully analyze the precedence of entries:

- If a value is not set in a <domain-config> entry or in a parent <domain-config>, the configurations in place will be based on the <base-config>
- If not defined in this entry, the default configurations will be used.

Take a look at this example of a Network Security Configuration for an app targeting Android 9 (API level 28):

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        <domain includeSubdomains="false">owasp.org</domain>
        <trust-anchors>
            <certificates src="system" />
            <certificates src="user" />
        </trust-anchors>
    </domain-config>
</network-security-config>
```

Some observations:

- There's no <base-config>, meaning that the default configuration for Android 9 (API level 28) or higher will be used for all other connections (only system CA will be trusted in principle).
- However, the <domain-config> overrides the default configuration allowing the app to trust both system and user CAs for the indicated <domain> (owasp.org).
- This doesn't affect subdomains because of includeSubdomains="false".

Putting all together we can translate the above Network Security Configuration to: "the app trusts system and user CAs for the owasp.org domain, excluding its subdomains. For any other domains the app will trust the system CAs only".

Reference

- owasp-mastg Testing Endpoint Identify Verification (MSTG-NETWORK-3) Static Analysis Analyzing Custom Trust Anchors

Rulebook

- *Custom trust anchor analysis (Required)*

5.3.2 Server Certificate Verification

5.3.2.1 Verification with TrustManager

TrustManager is a means of verifying conditions necessary for establishing a trusted connection in Android. The following conditions should be checked at this point:

- Has the certificate been signed by a trusted CA?
- Has the certificate expired?
- Is the certificate self-signed?

The following code snippet is sometimes used during development and will accept any certificate, overwriting the functions checkClientTrusted, checkServerTrusted, and getAcceptedIssuers. Such implementations should be avoided, and, if they are necessary, they should be clearly separated from production builds to avoid built-in security flaws.

```
TrustManager[] trustAllCerts = new TrustManager[] {
    new X509TrustManager() {
        @Override
        public X509Certificate[] getAcceptedIssuers() {
            return new java.security.cert.X509Certificate[] {};
        }

        @Override
        public void checkClientTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        }

        @Override
        public void checkServerTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        }
    }
 };

// SSLContext context
context.init(null, trustAllCerts, new SecureRandom());
```

Reference

- owasp-mastg Testing Endpoint Identify Verification (MSTG-NETWORK-3) Static Analysis Verifying the Server Certificate

Rulebook

- *Verification by TrustManager (Required)*

5.3.2.2 WebView Server Certificate Verification

Sometimes applications use a WebView to render the website associated with the application. This is true of HTML/JavaScript-based frameworks such as Apache Cordova, which uses an internal WebView for application interaction. When a WebView is used, the mobile browser performs the server certificate validation. Ignoring any TLS error that occurs when the WebView tries to connect to the remote website is a bad practice.

The following code will ignore TLS issues, exactly like the WebViewClient custom implementation provided to the WebView:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebViewClient(){
    @Override
    public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError␣
↪error) {
        //Ignore TLS certificate errors and instruct the WebViewClient to load the␣
↪website
        handler.proceed();
    }
});
```

Implementation of the Apache Cordova framework's internal WebView usage will ignore TLS errors in the method onReceivedSslError if the flag android:debuggable is enabled in the application manifest. Therefore, make sure that the app is not debuggable. See the test case "Testing If the App is Debuggable".

Reference

- owasp-mastg Testing Endpoint Identify Verification (MSTG-NETWORK-3) Static Analysis WebView Server Certificate Verification
- owasp-mastg Testing Endpoint Identify Verification (MSTG-NETWORK-3) Static Analysis Apache Cordova Certificate Verification

Rulebook

- *Bad Practices for Validating Server Certificates in WebView (Required)*

5.3.3 Hostname Verification

Another security flaw in client-side TLS implementations is the lack of hostname verification. Development environments usually use internal addresses instead of valid domain names, so developers often disable hostname verification (or force an application to allow any hostname) and simply forget to change it when their application goes to production. The following code disables hostname verification:

```
final static HostnameVerifier NO_VERIFY = new HostnameVerifier() {
    public boolean verify(String hostname, SSLSession session) {
        return true;
    }
};
```

With a built-in HostnameVerifier, accepting any hostname is possible:

```
HostnameVerifier NO_VERIFY = org.apache.http.conn.ssl.SSLSocketFactory
                             .ALLOW_ALL_HOSTNAME_VERIFIER;
```

Make sure that your application verifies a hostname before setting a trusted connection.

Reference

- owasp-mastg Testing Endpoint Identify Verification (MSTG-NETWORK-3) Static Analysis Hostname Verification

Rulebook

- *Hostname verification (Required)*

5.3.4 Rulebook

1. *MITM attack potential depending on target SDK version (Required)*
2. *Custom trust anchor analysis (Required)*
3. *Verification by TrustManager (Required)*
4. *Bad Practices for Validating Server Certificates in WebView (Required)*
5. *Hostname verification (Required)*

5.3.4.1 MITM attack potential depending on target SDK version (Required)

Applications targeting Android 7.0 (API level 24) or higher will use a default Network Security Configuration that doesn't trust any user supplied CAs, reducing the possibility of MITM attacks by luring users to install malicious CAs.

Decode the app using apktool and verify that the targetSdkVersion in apktool.yml is equal to or higher than 24.

If this is violated, the following may occur.

- Increased likelihood of MITM attack to install malicious CA.

5.3.4.2 Custom trust anchor analysis (Required)

Even with targetSdkVersion >=24, developers can use a custom network security configuration to disable the default protection and define a custom trust anchor to force the app to trust the CA provided by the user.

The android:networkSecurityConfig setting in AndroidManifest.xml should be checked.

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

The Network Security Configuration file set in android:networkSecurityConfig should be checked to verify the status of the following tags.

- <base-config>
- <trust-anchors>
- <certificates>

* <certificates src="user" > setting should be avoided.

Tags not set with a unique configuration inherit the setting at <base-config>, and if <base-config> is not set, the platform default is set.

You should carefully analyze the precedence of entries:

- If a value is not set in a <domain-config> entry or in a parent <domain-config>, the configurations in place will be based on the <base-config>
- If not defined in this entry, the default configurations will be used.

Take a look at this example of a Network Security Configuration for an app targeting Android 9 (API level 28):

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        <domain includeSubdomains="false">owasp.org</domain>
        <trust-anchors>
            <certificates src="system" />
            <certificates src="user" />
        </trust-anchors>
    </domain-config>
</network-security-config>
```

Some observations:

- There's no <base-config>, meaning that the default configuration for Android 9 (API level 28) or higher will be used for all other connections (only system CA will be trusted in principle).
- However, the <domain-config> overrides the default configuration allowing the app to trust both system and user CAs for the indicated <domain> (owasp.org).
- This doesn't affect subdomains because of includeSubdomains="false".

Putting all together we can translate the above Network Security Configuration to: "the app trusts system and user CAs for the owasp.org domain, excluding its subdomains. For any other domains the app will trust the system CAs only".

If this is violated, the following may occur.

- Increased likelihood of MITM attacks that force the installation of malicious CAs.

5.3.4.3 Verification by TrustManager (Required)

When the functions checkClientTrusted, checkServerTrusted, and getAcceptedIssuers are overridden using TrustManager, If all certificates are accepted without verifying client certificates, as in the sample code below, secure communication cannot be guaranteed. In the case of development, the following sample code is convenient for checking the operation with a self-certified certificate, but the process should be separated to prevent accidental incorporation into the production version.

```
TrustManager[] trustAllCerts = new TrustManager[] {
    new X509TrustManager() {
        @Override
        public X509Certificate[] getAcceptedIssuers() {
            return new java.security.cert.X509Certificate[] {};
        }

        @Override
        public void checkClientTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        }

        @Override
        public void checkServerTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        }
    }
 };

// SSLContext context
context.init(null, trustAllCerts, new SecureRandom());
```

The sample code below is the process of initializing TrustManager and setting HttpsURLConnection in order to trust a set of specific CAs.

```
    // Load CAs from an InputStream
    // (could be from a resource or ByteArrayInputStream or ...)
    CertificateFactory cf = CertificateFactory.getInstance("X.509");
    // From https://www.washington.edu/itconnect/security/ca/load-der.crt
    InputStream caInput = new BufferedInputStream(new FileInputStream("load-der.crt
↪"));
    Certificate ca;
    try {
        ca = cf.generateCertificate(caInput);
        System.out.println("ca=" + ((X509Certificate) ca).getSubjectDN());
    } finally {
        caInput.close();
    }

    // Create a KeyStore containing our trusted CAs
    String keyStoreType = KeyStore.getDefaultType();
    KeyStore keyStore = KeyStore.getInstance(keyStoreType);
    keyStore.load(null, null);
    keyStore.setCertificateEntry("ca", ca);

    // Create a TrustManager that trusts the CAs in our KeyStore
    String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
    TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
    tmf.init(keyStore);

    // Create an SSLContext that uses our TrustManager
    SSLContext context = SSLContext.getInstance("TLS");
    context.init(null, tmf.getTrustManagers(), null);

    // Tell the URLConnection to use a SocketFactory from our SSLContext
    URL url = new URL("https://certs.cac.washington.edu/CAtest/");
    HttpsURLConnection urlConnection =
        (HttpsURLConnection)url.openConnection();
    urlConnection.setSSLSocketFactory(context.getSocketFactory());
    InputStream in = urlConnection.getInputStream();
    copyInputStreamToOutputStream(in, System.out);
```

Reference

- Security with network protocols Unknown certificate authority

If this is violated, the following may occur.

- If verification with a self-certificate is included, it is not possible to determine if the certificate is trustworthy.

5.3.4.4 Bad Practices for Validating Server Certificates in WebView (Required)

Sometimes applications use a WebView to render the website associated with the application. This is true of HTML/JavaScript-based frameworks such as Apache Cordova, which uses an internal WebView for application interaction. When a WebView is used, the mobile browser performs the server certificate validation. Ignoring any TLS error that occurs when the WebView tries to connect to the remote website is a bad practice.

The sample code below is an example of how to ignore TLS errors and load a website into WebViewClient.

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebViewClient(){
    @Override
    public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError␣
↪error) {
        //Ignore TLS certificate errors and instruct the WebViewClient to load the␣
↪website
```

(continues on next page)

(continued from previous page)

```
        handler.proceed();
    }
});
```

Implementation of the Apache Cordova framework's internal WebView usage will ignore TLS errors in the method onReceivedSslError if the flag android:debuggable is enabled in the application manifest. Therefore, make sure that the app is not debuggable.

If this is violated, the following may occur.

- Vulnerable to man-in-the-middle attacks.

5.3.4.5 Hostname verification (Required)

During the development phase, the developer may have disabled hostname validation (or allowed arbitrary hostnames in the application). In some cases, the validation is disabled without making any changes when the production environment goes live.

The following is a case where this is disabled.

```
final static HostnameVerifier NO_VERIFY = new HostnameVerifier() {
    public boolean verify(String hostname, SSLSession session) {
        return true;
    }
};
```

The following is a list of arbitrary host names that are accepted.

```
HostnameVerifier NO_VERIFY = org.apache.http.conn.ssl.SSLSocketFactory
                             .ALLOW_ALL_HOSTNAME_VERIFIER;
```

Host name verification should be performed when connecting to the production environment.

If this is violated, the following may occur.

- It is possible to communicate with a host that is not a trusted destination host.

6 Platform Interaction Requirements

6.1 MSTG-PLATFORM-1

The app only requests the minimum set of permissions necessary.

6.1.1 Testing App Permissions

Android assigns a distinct system identity (Linux user ID and group ID) to every installed app. Because each Android app operates in a process sandbox, apps must explicitly request access to resources and data that are outside their sandbox. They request this access by declaring the permissions they need to use system data and features. Depending on how sensitive or critical the data or feature is, the Android system will grant the permission automatically or ask the user to approve the request.

Android permissions are classified into four different categories on the basis of the protection level they offer:

- Normal: This permission gives apps access to isolated application-level features with minimal risk to other apps, the user, and the system. For apps targeting Android 6.0 (API level 23) or higher, these permissions are granted automatically at installation time. For apps targeting a lower API level, the user needs to approve them at installation time. Example: android.permission.INTERNET.
- Dangerous: This permission usually gives the app control over user data or control over the device in a way that impacts the user. This type of permission may not be granted at installation time; whether the app should have the permission may be left for the user to decide. Example: android.permission.RECORD_AUDIO.
- Signature: This permission is granted only if the requesting app was signed with the same certificate used to sign the app that declared the permission. If the signature matches, the permission will be granted automatically. This permission is granted at installation time. Example: android.permission.ACCESS_MOCK_LOCATION.
- SystemOrSignature: This permission is granted only to applications embedded in the system image or signed with the same certificate used to sign the application that declared the permission. Example: android.permission.ACCESS_DOWNLOAD_MANAGER.

A list of all permissions can be found in the Android developer documentation as well as concrete steps on how to:

- Declare app permissions in your app' s manifest file.
- Request app permissions programmatically.
- Define a Custom App Permission to share your app resources and capabilities with other apps.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Overview

6.1.2 Changes to permissions per API level

6.1.2.1 Android 8.0 (API level 26) Changes

The following changes affect all apps running on Android 8.0 (API level 26), even to those apps targeting lower API levels.

- **Contacts provider usage stats change**: when an app requests the READ_CONTACTS permission, queries for contact's usage data will return approximations rather than exact values (the auto-complete API is not affected by this change).

Apps targeting Android 8.0 (API level 26) or higher are affected by the following:

- Account access and discoverability improvements: Apps can no longer get access to user accounts only by having the GET_ACCOUNTS permission granted, unless the authenticator owns the accounts or the user grants that access.
- New telephony permissions: the following permissions (classified as dangerous) are now part of the PHONE permissions group:
  - The ANSWER_PHONE_CALLS permission allows to answer incoming phone calls programmatically (via acceptRingingCall).
  - The READ_PHONE_NUMBERS permission grants read access to the phone numbers stored in the device.
- Restrictions when granting dangerous permissions: Dangerous permissions are classified into permission groups (e.g. the STORAGE group contains READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE). Before Android 8.0 (API level 26), it was sufficient to request one permission of the group in order to get all permissions of that group also granted at the same time. This has changed starting at Android 8.0 (API level 26): whenever an app requests a permission at runtime, the system will grant exclusively that specific permission. However, note that all subsequent requests for permissions in that permission group will be automatically granted without showing the permissions dialog to the user. See this example from the Android developer documentation:

Suppose an app lists both READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE in its manifest. The app requests READ_EXTERNAL_STORAGE and the user grants it. If the app targets API level 25 or lower, the system also grants WRITE_EXTERNAL_STORAGE at the same time, because it belongs to the same STORAGE permission group and is also registered in the manifest. If the app targets Android 8.0 (API level 26), the system grants only READ_EXTERNAL_STORAGE at that time; however, if the app later requests WRITE_EXTERNAL_STORAGE, the system immediately grants that privilege without prompting the user.

You can see the list of permission groups in the Android developer documentation. To make this a bit more confusing, Google also warns that particular permissions might be moved from one group to another in future versions of the Android SDK and therefore, the logic of the app shouldn't rely on the structure of these permission groups. The best practice is to explicitly request every permission whenever it's needed.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Android 8.0 (API level 26) Changes

Rulebook

- *Explicitly request all permissions when needed (Required)*

6.1.2.2 Android 9 (API Level 28) Changes

The following changes affect all apps running on Android 9, even to those apps targeting API levels lower than 28.

- Restricted access to call logs: READ_CALL_LOG, WRITE_CALL_LOG, and PROCESS_OUTGOING_CALLS (dangerous) permissions are moved from PHONE to the new CALL_LOG permission group. This means that being able to make phone calls (e.g. by having the permissions of the PHONE group granted) is not sufficient to get access to the call logs.
- Restricted access to phone numbers: apps wanting to read the phone number require the READ_CALL_LOG permission when running on Android 9 (API level 28).
- Restricted access to Wi-Fi location and connection information: SSID and BSSID values cannot be retrieved (e.g. via WifiManager.getConnectionInfo unless all of the following is true:
  - The ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION permission.
  - The ACCESS_WIFI_STATE permission.
  - Location services are enabled (under Settings -> Location).

Apps targeting Android 9 (API level 28) or higher are affected by the following:

- Build serial number deprecation: device's hardware serial number cannot be read (e.g. via Build.getSerial) unless the READ_PHONE_STATE (dangerous) permission is granted.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Android 9 (API Level 28) Changes

Rulebook

- *Explicitly request all permissions when needed (Required)*

6.1.2.3 Android 10 (API level 29) Changes

Android 10 (API level 29) introduces several user privacy enhancements. The changes regarding permissions affect to all apps running on Android 10 (API level 29), including those targeting lower API levels.

- Restricted Location access: new permission option for location access "only while using the app".
- Scoped storage by default: apps targeting Android 10 (API level 29) don't need to declare any storage permission to access their files in the app specific directory in external storage as well as for files creates from the media store.
- Restricted access to screen contents: READ_FRAME_BUFFER, CAPTURE_VIDEO_OUTPUT, and CAPTURE_SECURE_VIDEO_OUTPUT permissions are now signature-access only, which prevents silent access to the device's screen contents.
- User-facing permission check on legacy apps: when running an app targeting Android 5.1 (API level 22) or lower for the first time, users will be prompted with a permissions screen where they can revoke access to specific legacy permissions (which previously would be automatically granted at installation time).

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Android 10 (API level 29) Changes

Rulebook

- *Explicitly request all permissions when needed (Required)*

6.1.3 Integration with other app components

6.1.3.1 Activity Permission Enforcement

Permissions are applied via android:permission attribute within the <activity> tag in the manifest. These permissions restrict which applications can start that Activity. The permission is checked during Context.startActivity and Activity.startActivityForResult. Not holding the required permission results in a SecurityException being thrown from the call.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Activity Permission Enforcement

6.1.3.2 Service Permission Enforcement

Permissions applied via android:permission attribute within the <service> tag in the manifest restrict who can start or bind to the associated Service. The permission is checked during Context.startService, Context.stopService and Context.bindService. Not holding the required permission results in a SecurityException being thrown from the call.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Service Permission Enforcement

6.1.3.3 Broadcast Permission Enforcement

Permissions applied via android:permission attribute within the <receiver> tag restrict access to send broadcasts to the associated BroadcastReceiver. The held permissions are checked after Context.sendBroadcast returns, while trying to deliver the sent broadcast to the given receiver. Not holding the required permissions doesn't throw an exception, the result is an unsent broadcast.

A permission can be supplied to Context.registerReceiver to control who can broadcast to a programmatically registered receiver. Going the other way, a permission can be supplied when calling Context.sendBroadcast to restrict which broadcast receivers are allowed to receive the broadcast.

Note that both a receiver and a broadcaster can require a permission. When this happens, both permission checks must pass for the intent to be delivered to the associated target. For more information, please reference the section "Restricting broadcasts with permissions" in the Android Developers Documentation.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Broadcast Permission Enforcement

Rulebook

- *Note that authorization may be required on both the receiving and sending sides of Broadcast (Required)*

6.1.3.4 Content Provider Permission Enforcement

Permissions applied via android:permission attribute within the <provider> tag restrict access to data in a ContentProvider. Content providers have an important additional security facility called URI permissions which is described next. Unlike the other components, ContentProviders have two separate permission attributes that can be set, android:readPermission restricts who can read from the provider, and android:writePermission restricts who can write to it. If a ContentProvider is protected with both read and write permissions, holding only the write permission does not also grant read permissions.

Permissions are checked when you first retrieve a provider and as operations are performed using the ContentProvider. Using ContentResolver.query requires holding the read permission; using ContentResolver.insert, ContentResolver.update, ContentResolver.delete requires the write permission. A SecurityException will be thrown from the call if proper permissions are not held in all these cases.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Content Provider Permission Enforcement

6.1.4 Content Provider URI Permissions

The standard permission system is not sufficient when being used with content providers. For example a content provider may want to limit permissions to READ permissions in order to protect itself, while using custom URIs to retrieve information. An application should only have the permission for that specific URI.

The solution is per-URI permissions. When starting or returning a result from an activity, the method can set Intent.FLAG_GRANT_READ_URI_PERMISSION and/or Intent.FLAG_GRANT_WRITE_URI_PERMISSION. This grants permission to the activity for the specific URI regardless if it has permissions to access to data from the content provider.

This allows a common capability-style model where user interaction drives ad-hoc granting of fine-grained permission. This can be a key facility for reducing the permissions needed by apps to only those directly related to their behavior. Without this model in place malicious users may access other member's email attachments or harvest contact lists for future use via unprotected URIs. In the manifest the android:grantUriPermissions attribute or the node help restrict the URIs.

Documentation for URI Permissions

- grantUriPermission
- revokeUriPermission
- checkUriPermission

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Content Provider URI Permissions
- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Documentation for URI Permissions

Rulebook

- *Applications should only have permissions to specific URIs (Required)*

6.1.5 Custom Permissions

Android allows apps to expose their services/components to other apps. Custom permissions are required for app access to the exposed components. You can define custom permissions in AndroidManifest.xml by creating a permission tag with two mandatory attributes: android:name and android:protectionLevel.

It is crucial to create custom permissions that adhere to the Principle of Least Privilege: permission should be defined explicitly for its purpose, with a meaningful and accurate label and description.

Below is an example of a custom permission called START_MAIN_ACTIVITY, which is required when launching the TEST_ACTIVITY Activity.

The first code block defines the new permission, which is self-explanatory. The label tag is a summary of the permission, and the description is a more detailed version of the summary. You can set the protection level according to the types of permissions that will be granted. Once you've defined your permission, you can enforce it by adding it to the application's manifest. In our example, the second block represents the component that we are going to restrict with the permission we created. It can be enforced by adding the android:permission attributes.

```
<permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY"
        android:label="Start Activity in myapp"
        android:description="Allow the app to launch the activity of myapp app,␣
↪any app you grant this permission will be able to launch main activity by myapp␣
↪app."
        android:protectionLevel="normal" />

<activity android:name="TEST_ACTIVITY"
    android:permission="com.example.myapp.permission.START_MAIN_ACTIVITY">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
```

(continues on next page)

(continued from previous page)

```
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Once the permission START_MAIN_ACTIVITY has been created, apps can request it via the uses-permission tag in the AndroidManifest.xml file. Any application granted the custom permission START_MAIN_ACTIVITY can then launch the TEST_ACTIVITY. Please note <uses-permission android:name=" myapp.permission.START_MAIN_ACTIVITY" /> must be declared before the <application> or an exception will occur at runtime. Please see the example below that is based on the permission overview and manifest-intro.

```
<manifest>
<uses-permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY" />
        <application>
            <activity>
            </activity>
        </application>
</manifest>
```

We recommend using a reverse-domain annotation when registering a permission, as in the example above (e.g. com.domain.application.permission) in order to avoid collisions with other applications.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Custom Permissions

Rulebook

- *Custom permissions are created according to the "Principle of Least Privilege" (Required)*
- *Register custom permissions with reverse-domain annotation (Recommended)*

6.1.6 Static Analysis

6.1.6.1 Android Permissions

Check permissions to make sure that the app really needs them and remove unnecessary permissions. For example, the INTERNET permission in the AndroidManifest.xml file is necessary for an Activity to load a web page into a WebView. Because a user can revoke an application' s right to use a dangerous permission, the developer should check whether the application has the appropriate permission each time an action is performed that would require that permission.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Go through the permissions with the developer to identify the purpose of every permission set and remove unnecessary permissions.

Besides going through the AndroidManifest.xml file manually, you can also use the Android Asset Packaging tool (aapt) to examine the permissions of an APK file.

aapt comes with the Android SDK within the build-tools folder. It requires an APK file as input. You may list the APKs in the device by running adb shell pm list packages -f | grep -i <keyword> as seen in "Listing Installed Apps" .

```
$ aapt d permissions app-x86-debug.apk
package: sg.vp.owasp_mobile.omtg_android
uses-permission: name='android.permission.WRITE_EXTERNAL_STORAGE'
uses-permission: name='android.permission.INTERNET'
```

Alternatively you may obtain a more detailed list of permissions via adb and the dumpsys tool:

```
$ adb shell dumpsys package sg.vp.owasp_mobile.omtg_android | grep permission
    requested permissions:
      android.permission.WRITE_EXTERNAL_STORAGE
      android.permission.INTERNET
      android.permission.READ_EXTERNAL_STORAGE
    install permissions:
      android.permission.INTERNET: granted=true
      runtime permissions:
```

Please reference this permissions overview for descriptions of the listed permissions that are considered dangerous.

```
READ_CALENDAR
WRITE_CALENDAR
READ_CALL_LOG
WRITE_CALL_LOG
PROCESS_OUTGOING_CALLS
CAMERA
READ_CONTACTS
WRITE_CONTACTS
GET_ACCOUNTS
ACCESS_FINE_LOCATION
ACCESS_COARSE_LOCATION
RECORD_AUDIO
READ_PHONE_STATE
READ_PHONE_NUMBERS
CALL_PHONE
ANSWER_PHONE_CALLS
ADD_VOICEMAIL
USE_SIP
BODY_SENSORS
SEND_SMS
RECEIVE_SMS
READ_SMS
RECEIVE_WAP_PUSH
RECEIVE_MMS
READ_EXTERNAL_STORAGE
WRITE_EXTERNAL_STORAGE
```

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Android Permissions

Rulebook

- *Check if permissions are really necessary for the app and remove unnecessary permissions (Required)*

6.1.6.2 Custom Permissions

Apart from enforcing custom permissions via the application manifest file, you can also check permissions programmatically. This is not recommended, however, because it is more error-prone and can be bypassed more easily with, e.g., runtime instrumentation. It is recommended that the ContextCompat.checkSelfPermission method is called to check if an activity has a specified permission. Whenever you see code like the following snippet, make sure that the same permissions are enforced in the manifest file.

```
private static final String TAG = "LOG";
int canProcess = checkCallingOrSelfPermission("com.example.perm.READ_INCOMING_MSG
↪");
if (canProcess != PERMISSION_GRANTED)
throw new SecurityException();
```

Or with ContextCompat.checkSelfPermission which compares it to the manifest file.

```
if (ContextCompat.checkSelfPermission(secureActivity.this, Manifest.READ_INCOMING_
↪MSG)
        != PackageManager.PERMISSION_GRANTED) {
            //!= stands for not equals PERMISSION_GRANTED
            Log.v(TAG, "Permission denied");
        }
```

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Android Permissions

Rulebook

- *Programmatic checking of custom permissions (Recommended)*

6.1.7 Requesting Permissions

If your application has permissions that need to be requested at runtime, the application must call the requestPermissions method in order to obtain them. The app passes the permissions needed and an integer request code you have specified to the user asynchronously, returning once the user chooses to accept or deny the request in the same thread. After the response is returned the same request code is passed to the app's callback method.

```
private static final String TAG = "LOG";
// We start by checking the permission of the current Activity
if (ContextCompat.checkSelfPermission(secureActivity.this,
        Manifest.permission.WRITE_EXTERNAL_STORAGE)
        != PackageManager.PERMISSION_GRANTED) {

    // Permission is not granted
    // Should we show an explanation?
    if (ActivityCompat.shouldShowRequestPermissionRationale(secureActivity.this,
        //Gets whether you should show UI with rationale for requesting permission.
        //You should do this only if you do not have permission and the permission␣
↪requested rationale is not communicated clearly to the user.
            Manifest.permission.WRITE_EXTERNAL_STORAGE)) {
        // Asynchronous thread waits for the users response.
        // After the user sees the explanation try requesting the permission again.
    } else {
        // Request a permission that doesn't need to be explained.
        ActivityCompat.requestPermissions(secureActivity.this,
                new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE},
                MY_PERMISSIONS_REQUEST_WRITE_EXTERNAL_STORAGE);
        // MY_PERMISSIONS_REQUEST_WRITE_EXTERNAL_STORAGE will be the app-defined␣
↪int constant.
        // The callback method gets the result of the request.
    }
} else {
    // Permission already granted debug message printed in terminal.
    Log.v(TAG, "Permission already granted.");
}
```

Please note that if you need to provide any information or explanation to the user it needs to be done before the call to requestPermissions, since the system dialog box can not be altered once called.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Requesting Permissions

Rulebook

- *Explicitly request all permissions when needed (Required)*

6.1.8 Handling Responses to Permission Requests

Now your app has to override the system method onRequestPermissionsResult to see if the permission was granted. This method receives the requestCode integer as input parameter (which is the same request code that was created in requestPermissions).

The following callback method may be used for WRITE_EXTERNAL_STORAGE.

```
@Override //Needed to override system method onRequestPermissionsResult()
public void onRequestPermissionsResult(int requestCode, //requestCode is what you␣
↪specified in requestPermissions()
        String permissions[], int[] permissionResults) {
    switch (requestCode) {
        case MY_PERMISSIONS_WRITE_EXTERNAL_STORAGE: {
            if (grantResults.length > 0
                && permissionResults[0] == PackageManager.PERMISSION_GRANTED) {
                // 0 is a canceled request, if int array equals requestCode␣
↪permission is granted.
            } else {
                // permission denied code goes here.
                Log.v(TAG, "Permission denied");
            }
            return;
        }
        // Other switch cases can be added here for multiple permission checks.
    }
}
```

Permissions should be explicitly requested for every needed permission, even if a similar permission from the same group has already been requested. For applications targeting Android 7.1 (API level 25) and older, Android will automatically give an application all the permissions from a permission group, if the user grants one of the requested permissions of that group. Starting with Android 8.0 (API level 26), permissions will still automatically be granted if a user has already granted a permission from the same permission group, but the application still needs to explicitly request the permission. In this case, the onRequestPermissionsResult handler will automatically be triggered without any user interaction.

For example if both READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE are listed in the Android Manifest but only permissions are granted for READ_EXTERNAL_STORAGE, then requesting WRITE_EXTERNAL_STORAGE will automatically have permissions without user interaction because they are in the same group and not explicitly requested.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Handling Responses to Permission Requests

Rulebook

- *Explicitly request all permissions when needed (Required)*

6.1.9 Permission Analysis

Always check whether the application is requesting permissions it actually requires. Make sure that no permissions are requested which are not related to the goal of the app, especially DANGEROUS and SIGNATURE permissions, since they can affect both the user and the application if mishandled. For instance, it should be suspicious if a single-player game app requires access to android.permission.WRITE_SMS.

When analyzing permissions, you should investigate the concrete use case scenarios of the app and always check if there are replacement APIs for any DANGEROUS permissions in use. A good example is the SMS Retriever API which streamlines the usage of SMS permissions when performing SMS-based user verification. By using this API an application does not have to declare DANGEROUS permissions which is a benefit to both the user and developers of the application, who doesn't have to submit the Permissions Declaration Form.

Reference

- owasp mastg Testing App Permissions (MSTG-PLATFORM-1) Permission Analysis

Rulebook

- *Check if permissions are really necessary for the app and remove unnecessary permissions (Required)*

6.1.10 Dynamic Analysis

Permissions for installed applications can be retrieved with adb. The following extract demonstrates how to examine the permissions used by an application.

```
$ adb shell dumpsys package com.google.android.youtube
...
declared permissions:
  com.google.android.youtube.permission.C2D_MESSAGE: prot=signature, INSTALLED
requested permissions:
  android.permission.INTERNET
  android.permission.ACCESS_NETWORK_STATE
install permissions:
  com.google.android.c2dm.permission.RECEIVE: granted=true
  android.permission.USE_CREDENTIALS: granted=true
  com.google.android.providers.gsf.permission.READ_GSERVICES: granted=true
...
```

The output shows all permissions using the following categories:

- declared permissions: list of all custom permissions.
- requested and install permissions: list of all install-time permissions including normal and signature permissions.
- runtime permissions: list of all dangerous permissions.

When doing the dynamic analysis:

- Evaluate whether the app really needs the requested permissions. For instance: a single-player game that requires access to android.permission.WRITE_SMS, might not be a good idea.
- In many cases the app could opt for alternatives to declaring permissions, such as:
  - requesting the ACCESS_COARSE_LOCATION permission instead of ACCESS_FINE_LOCATION. Or even better not requesting the permission at all, and instead ask the user to enter a postal code.
  - invoking the AC*TION_IMAGE_CAPTURE or ACTION_VIDEO_CAPTURE intent action instead of requesting the CAMERA permission.
  - using Companion Device Pairing (Android 8.0 (API level 26) and higher) when pairing with a Bluetooth device instead of declaring the ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATIION, or BLUETOOTH_ADMIN permissions.
- Use the Privacy Dashboard (Android 12 (API level 31) and higher) to verify how the app explains access to sensitive information. To obtain detail about a specific permission you can refer to the Android Documentation.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Dynamic Analysis

Rulebook

- *Check if permissions are really necessary for the app and remove unnecessary permissions (Required)*

6.1.11 Rulebook

1. *Explicitly request all permissions when needed (Required)*
2. *Note that authorization may be required on both the receiving and sending sides of Broadcast (Required)*
3. *Applications should only have permissions to specific URIs (Required)*
4. *Custom permissions are created according to the "Principle of Least Privilege" (Required)*
5. *Register custom permissions with reverse-domain annotation (Recommended)*
6. *Check if permissions are really necessary for the app and remove unnecessary permissions (Required)*
7. *Programmatic checking of custom permissions (Recommended)*

6.1.11.1 Explicitly request all permissions when needed (Required)

Google warns that certain permissions may be moved from one group to another in future versions of the Android SDK. They warn that the app logic should not rely on the structure of these permission groups, as they may move from one group to another. Therefore, the best practice is to explicitly request all permissions when needed.

Below is a sample code for explicitly acquiring write permission to external storage.

Declare permission to write to external storage in AndroidManifest.xml.

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Sample code to request permissions when needed.

```
private static final String TAG = "LOG";
// We start by checking the permission of the current Activity
if (ContextCompat.checkSelfPermission(secureActivity.this,
        Manifest.permission.WRITE_EXTERNAL_STORAGE)
        != PackageManager.PERMISSION_GRANTED) {

    // Permission is not granted
    // Should we show an explanation?
    if (ActivityCompat.shouldShowRequestPermissionRationale(secureActivity.this,
        //Gets whether you should show UI with rationale for requesting permission.
        //You should do this only if you do not have permission and the permission␣
↪requested rationale is not communicated clearly to the user.
            Manifest.permission.WRITE_EXTERNAL_STORAGE)) {
        // Asynchronous thread waits for the users response.
        // After the user sees the explanation try requesting the permission again.
    } else {
        // Request a permission that doesn't need to be explained.
        ActivityCompat.requestPermissions(secureActivity.this,
                new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE},
                MY_PERMISSIONS_REQUEST_WRITE_EXTERNAL_STORAGE);
        // MY_PERMISSIONS_REQUEST_WRITE_EXTERNAL_STORAGE will be the app-defined␣
↪int constant.
        // The callback method gets the result of the request.
    }
} else {
    // Permission already granted debug message printed in terminal.
    Log.v(TAG, "Permission already granted.");
}
```

Sample code for permission request response.

```
@Override //Needed to override system method onRequestPermissionsResult()
public void onRequestPermissionsResult(int requestCode, //requestCode is what you␣
↪specified in requestPermissions()
```

(continues on next page)

(continued from previous page)

```
        String permissions[], int[] permissionResults) {
    switch (requestCode) {
        case MY_PERMISSIONS_WRITE_EXTERNAL_STORAGE: {
            if (grantResults.length > 0
                && permissionResults[0] == PackageManager.PERMISSION_GRANTED) {
                // 0 is a canceled request, if int array equals requestCode␣
↪permission is granted.
            } else {
                // permission denied code goes here.
                Log.v(TAG, "Permission denied");
            }
            return;
        }
        // Other switch cases can be added here for multiple permission checks.
    }
}
```

If this is violated, the following may occur.

- App logic needs to be reviewed if certain permissions are moved from one group to another in future versions of the Android SDK.

6.1.11.2 Note that authorization may be required on both the receiving and sending sides of Broadcast (Required)

Permissions can be set to restrict Broadcast to a set of apps with certain permissions. Restrictions can be applied to the sender or receiver of the Broadcast. Note that this requires privileges on both the receiving and sending sides.

Transmission with set permissions

When calling sendBroadcast(Intent, String) or sendOrderedBroadcast(Intent, String, BroadcastReceiver, Handler, int, String, Bundle), you can specify permission parameters. Only receivers that have requested permission using tags in the manifest (and receivers that have been granted permission in conjunction for security reasons) may receive Broadcasts.

The sample code below is an example of sending Broadcast with the permission parameter specified.

sendBroadcast

```
sendBroadcast(Intent("com.example.NOTIFY"), Manifest.permission.SEND_SMS)
```

sendOrderedBroadcast

To receive Broadcast, the receiving application must request authorization as shown in the sample code below.

```
sendOrderedBroadcast(Intent("com.example.NOTIFY"), Manifest.permission.SEND_SMS,␣
↪new BroadcastReceiver() {
            @SuppressLint("NewApi")
                        @Override
            public void onReceive(Context context, Intent intent) {
                Bundle results = getResultExtras(true);
            }
        }, null, Activity.RESULT_OK, null, null);
```

You may specify an existing system authority, such as SEND_SMS, or you may define a custom authority using the <permission > element to define custom permissions.

Receiving with set permissions

If the authorization parameter is specified when registering BroadcastReceiver (registerReceiver(BroadcastReceiver, IntentFilter, String, Handler) or <receiver> tag) in Manifest Broadcast that requested authorization using the <uses-permission> tag. Only the sender (and also the sender that has been granted permission in conjunction for security reasons) can send intent to the Receiver.

Assume that the receiving application has a Receiver declared in Manifest as shown below.

```
<receiver android:name=".MyBroadcastReceiver"
              android:permission="android.permission.SEND_SMS">
        <intent-filter>
            <action android:name="android.intent.action.AIRPLANE_MODE"/>
        </intent-filter>
    </receiver>
```

Alternatively, assume that there is a Receiver registered with the context in the receiving application.

```
var filter = IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED)
registerReceiver(receiver, filter, Manifest.permission.SEND_SMS, null )
```

To be able to send Broadcast to the above Receiver, it is necessary to request authorization from the sending application, as shown in the sample code below.

```
<uses-permission android:name="android.permission.SEND_SMS"/>
```

If this is violated, the following may occur.

- Broadcast targets cannot be restricted.

6.1.11.3 Applications should only have permissions to specific URIs (Required)

A ContentProvider may want to limit permissions to READ permissions to protect itself while using a custom URI to retrieve information. The application should only have permissions to that specific URI.

The solution is per-URI permissions. When starting an activity or returning a result, a method may set both or either Intent.FLAG_GRANT_READ_URI_PERMISSION and Intent.FLAG_GRANT_WRITE_URI_PERMISSION. FLAG_GRANT_WRITE_URI_PERMISSION and Intent. This will grant permission to the activity of a particular URI, regardless of whether or not it has permission to access data from ContentProvider.

The following is an example of specifying and granting permissions for each URI.

The sample code below is a declaration in AndroidManifest.xml for granting permissions.

```
<provider android:name=".MyProvider"
          android:authorities="com.example.sampleprovider.myprovider"
          android:grantUriPermissions="true" />
```

The sample code below is an example of processing for granting permissions and terminating permissions.

```
// Allowed by grantUriPermission
grantUriPermission("com.example.sampleresolver",
                   Uri.parse("content://com.example.sampleprovider/myprovider/"),
                   Intent.FLAG_GRANT_READ_URI_PERMISSION);

// Subsequent accesses only allow reading (query)

// Revoke UriPermission with revokeUriPermission
revokeUriPermission(Uri.parse("content://com.example.sampleprovider/myprovider/"),
                    Intent.FLAG_GRANT_READ_URI_PERMISSION);
```

This would allow for a general capability-style model in which fine-grained permissions are granted on an ad hoc basis depending on user interaction. This could be an important feature to reduce the number of permissions needed by an app to only those directly related to the app' s operation. Without this model, a malicious user could access other members' email attachments or retrieve contact lists for future use via unprotected URIs.

If this is violated, the following may occur.

- Unable to restrict access to ContentProvider from unintended apps.

6.1.11.4 Custom permissions are created according to the "Principle of Least Privilege" (Required)

It is important that custom permissions be created in accordance with the Principle of Least Privilege. Permissions should be explicitly defined with meaningful and precise labels and descriptions according to their purpose.

The following is an example of explicitly specifying custom permissions and using them in other applications.

The following is an example declaration of a custom permission called "START_MAIN_ACTIVITY" .

```
<permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY"
        android:label="Start Activity in myapp"
        android:description="Allow the app to launch the activity of myapp app,␣
↪any app you grant this permission will be able to launch main activity by myapp␣
↪app."
        android:protectionLevel="normal" />

<activity android:name="TEST_ACTIVITY"
    android:permission="com.example.myapp.permission.START_MAIN_ACTIVITY">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
     </intent-filter>
</activity>
```

The following is a declaration to use the created "START_MAIN_ACTIVITY" in other applications.

```
<manifest>
<uses-permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY" />
        <application>
            <activity>
            </activity>
        </application>
</manifest>
```

If this is violated, the following may occur.

- Unintended security breaches may occur in the event of an anomaly.

6.1.11.5 Register custom permissions with reverse-domain annotation (Recommended)

It is recommended that custom permissions be registered with reverse-domain annotation.

The following is an example of registering custom permissions with reverse-domain annotation when the domain is "example.com" .

```
<permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY"
        android:label="Start Activity in myapp"
        android:description="Allow the app to launch the activity of myapp app,␣
↪any app you grant this permission will be able to launch main activity by myapp␣
↪app."
        android:protectionLevel="normal" />
```

If this is not noted, the following may occur.

- Registering custom permission names can conflict with other apps.

6.1.11.6 Check if permissions are really necessary for the app and remove unnecessary permissions (Required)

Because users can revoke the rights of applications that use DANGEROUS permissions, developers should check to see that the application has the proper permissions each time an action requiring those permissions is performed.

Review the permissions with the developer, identify the purpose of all permissions, and remove any unnecessary permissions.

Below is the declaration of permissions in AndroidManifest.xml.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Analyze permissions.

Also, by analyzing permissions and investigating specific use case scenarios for your app, you should always check to see if there is an API that can replace the DANGEROUS permissions you are using. A good example is the SMS Retriever API, which streamlines the use of SMS permissions when performing SMS-based user authentication. By using this API, applications do not need to declare DANGEROUS permissions, and both the user and the application developer can use the Permissions Declaration Form for both the user and the application developer.

One method of analysis is the Privacy Dashboard (Android 12 (API level 31) and later) to see how apps describe their access to sensitive information.

If this is violated, the following may occur.

- The presence of unnecessary DANGEROUS permissions requires unnecessary permission request logic for users.

6.1.11.7 Programmatic checking of custom permissions (Recommended)

Aside from enforcing custom permissions through the application manifest file, it is also possible to check permissions programmatically.

As one method, we recommend the ContextCompat.checkSelfPermission method.

The sample code below shows how to check custom permissions using the ContextCompat.checkSelfPermission method.

```
if (ContextCompat.checkSelfPermission(secureActivity.this, Manifest.READ_INCOMING_
↪MSG)
        != PackageManager.PERMISSION_GRANTED) {
            //!= stands for not equals PERMISSION_GRANTED
            Log.v(TAG, "Permission denied");
        }
```

Alternatively, the checkCallingOrSelfPermission method can be used. This method is not recommended because it is error prone and can be easily bypassed by runtime instrumentation.

The sample code below shows how to check custom permissions using the checkCallingOrSelfPermission method.

```
private static final String TAG = "LOG";
int canProcess = checkCallingOrSelfPermission("com.example.perm.READ_INCOMING_MSG
↪");
if (canProcess != PERMISSION_GRANTED)
throw new SecurityException();
```

When performing such a programmatic checking process, make sure that the same permissions are enforced in the manifest file.

If this is not noted, the following may occur.

- The presence of unnecessary DANGEROUS permissions requires unnecessary permission request logic for users.

6.2 MSTG-PLATFORM-2

All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources.

6.2.1 Cross-Site Scripting Flaws

Cross-site scripting (XSS) issues allow attackers to inject client-side scripts into web pages viewed by users. This type of vulnerability is prevalent in web applications. When a user views the injected script in a browser, the attacker gains the ability to bypass the same origin policy, enabling a wide variety of exploits (e.g. stealing session cookies, logging key presses, performing arbitrary actions, etc.).

In the context of native apps, XSS risks are far less prevalent for the simple reason these kinds of applications do not rely on a web browser. However, apps using WebView components, such as WKWebView or the deprecated UIWebView on iOS and WebView on Android, are potentially vulnerable to such attacks.

An older but well-known example is the local XSS issue in the Skype app for iOS, first identified by Phil Purviance. The Skype app failed to properly encode the name of the message sender, allowing an attacker to inject malicious JavaScript to be executed when a user views the message. In his proof-of-concept, Phil showed how to exploit the issue and steal a user's address book.

Reference

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2)

6.2.1.1 Static Analysis

Take a close look at any WebViews present and investigate for untrusted input rendered by the app.

XSS issues may exist if the URL opened by WebView is partially determined by user input. The following example is from an XSS issue in the Zoho Web Service, reported by Linus Särud.

Java

```
webView.loadUrl("javascript:initialize(" + myNumber + ");");
```

Kotlin

```
webView.loadUrl("javascript:initialize($myNumber);")
```

Another example of XSS issues determined by user input is public overridden methods.

Java

```
@Override
public boolean shouldOverrideUrlLoading(WebView view, String url) {
  if (url.substring(0,6).equalsIgnoreCase("yourscheme:")) {
    // parse the URL object and execute functions
  }
}
```

Kotlin

```
    fun shouldOverrideUrlLoading(view: WebView, url: String): Boolean {
        if (url.substring(0, 6).equals("yourscheme:", ignoreCase = true)) {
            // parse the URL object and execute functions
        }
    }
```

Sergey Bobrov was able to take advantage of this in the following HackerOne report. Any input to the HTML parameter would be trusted in Quora's ActionBarContentActivity. Payloads were successful using adb, clipboard data via ModalContentActivity, and Intents from 3rd party applications.

- ADB

```
$ adb shell
$ am start -n com.quora.android/com.quora.android.ActionBarContentActivity \
-e url 'http://test/test' -e html 'XSS<script>alert(123)</script>'
```

- Clipboard Data

```
$ am start -n com.quora.android/com.quora.android.ModalContentActivity  \
-e url 'http://test/test' -e html \
'<script>alert(QuoraAndroid.getClipboardData());</script>'
```

- 3rd party Intent in Java or Kotlin :

```
Intent i = new Intent();
i.setComponent(new ComponentName("com.quora.android",
"com.quora.android.ActionBarContentActivity"));
i.putExtra("url","http://test/test");
i.putExtra("html","XSS PoC <script>alert(123)</script>");
view.getContext().startActivity(i);
```

```
val i = Intent()
i.component = ComponentName("com.quora.android",
"com.quora.android.ActionBarContentActivity")
i.putExtra("url", "http://test/test")
i.putExtra("html", "XSS PoC <script>alert(123)</script>")
view.context.startActivity(i)
```

If a WebView is used to display a remote website, the burden of escaping HTML shifts to the server side. If an XSS flaw exists on the web server, this can be used to execute script in the context of the WebView. As such, it is important to perform static analysis of the web application source code.

Verify that the following best practices have been followed:

- No untrusted data is rendered in HTML, JavaScript or other interpreted contexts unless it is absolutely necessary.
- Appropriate encoding is applied to escape characters, such as HTML entity encoding. Note: escaping rules become complicated when HTML is nested within other code, for example, rendering a URL located inside a JavaScript block.

Consider how data will be rendered in a response. For example, if data is rendered in a HTML context, six control characters that must be escaped:

Table 6.2.1.1.1 List of control characters that must be escaped

Character	Escaped
&	&amp;
<	&lt;
>	&gt;
“	&quot;
‘	&#x27;
/	&#x2F;

For a comprehensive list of escaping rules and other prevention measures, refer to the OWASP XSS Prevention Cheat Sheet.

Reference

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2) Static Analysis

Rulebook

- *Check WebView and investigate for unreliable input rendered by the app (Required)*

6.2.1.2 Dynamic Analysis

XSS issues can be best detected using manual and/or automated input fuzzing, i.e. injecting HTML tags and special characters into all available input fields to verify the web application denies invalid inputs or escapes the HTML meta-characters in its output.

A reflected XSS attack refers to an exploit where malicious code is injected via a malicious link. To test for these attacks, automated input fuzzing is considered to be an effective method. For example, the BURP Scanner is highly effective in identifying reflected XSS vulnerabilities. As always with automated analysis, ensure all input vectors are covered with a manual review of testing parameters.

Reference

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2) Dynamic Analysis

6.2.2 Using Data Storage

6.2.2.1 Using Shared Preferences

When you use the SharedPreferences.Editor to read or write int/boolean/long values, you cannot check whether the data is overridden or not. However: it can hardly be used for actual attacks other than chaining the values (e.g. no additional exploits can be packed which will take over the control flow). In the case of a String or a StringSet you should be careful with how the data is interpreted. Using reflection based persistence? Check the section on "Testing Object Persistence" for Android to see how it should be validated. Using the SharedPreferences.Editor to store and read certificates or keys? Make sure you have patched your security provider given vulnerabilities such as found in Bouncy Castle.

In all cases, having the content HMACed can help to ensure that no additions and/or changes have been applied.

Reference

- owasp-mastg Testing Local Storage for Input Validation (MSTG-PLATFORM-2) Using Shared Preferences

Rulebook

- *Notes on input validation when using SharedPreferences (Required)*

6.2.2.2 Using Other Storage Mechanisms

In case other public storage mechanisms (than the SharedPreferences.Editor) are used, the data needs to be validated the moment it is read from the storage mechanism.

Reference

- owasp-mastg Testing Local Storage for Input Validation (MSTG-PLATFORM-2) Using Other Storage Mechanisms

Rulebook

- *Notes on input validation when using public storage mechanisms other than SharedPreferences (Required)*

6.2.3 Testing for Injection Flaws

Android apps can expose functionality through deep links (which are a part of Intents). They can expose functionality to:

- other apps (via deep links or other IPC mechanisms, such as Intents or BroadcastReceivers).
- the user (via the user interface).

None of the input from these sources can be trusted; it must be validated and/or sanitized. Validation ensures processing of data that the app is expecting only. If validation is not enforced, any input can be sent to the app, which may allow an attacker or malicious app to exploit app functionality.

The following portions of the source code should be checked if any app functionality has been exposed:

- Deep Links. Check the test case "Testing Deep Links" as well for further test scenarios.
- IPC Mechanisms (Intents, Binders, Android Shared Memory, or BroadcastReceivers). Check the test case "Testing for Sensitive Functionality Exposure Through IPC" as well for further test scenarios.
- User interface. Check the test case "Testing for Overlay Attacks" .

An example of a vulnerable IPC mechanism is shown below.

You can use ContentProviders to access database information, and you can probe services to see if they return data. If data is not validated properly, the content provider may be prone to SQL injection while other apps are interacting with it. See the following vulnerable implementation of a ContentProvider.

```
<provider
    android:name=".OMTG_CODING_003_SQL_Injection_Content_Provider_Implementation"
    android:authorities="sg.vp.owasp_mobile.provider.College">
</provider>
```

The AndroidManifest.xml above defines a content provider that's exported and therefore available to all other apps. The query function in the OMTG_CODING_003_SQL_Injection_Content_Provider_Implementation.java class should be inspected.

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,String[]␣
↪selectionArgs, String sortOrder) {
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
    qb.setTables(STUDENTS_TABLE_NAME);

    switch (uriMatcher.match(uri)) {
        case STUDENTS:
            qb.setProjectionMap(STUDENTS_PROJECTION_MAP);
            break;

        case STUDENT_ID:
            // SQL Injection when providing an ID
            qb.appendWhere( _ID + "=" + uri.getPathSegments().get(1));
            Log.e("appendWhere",uri.getPathSegments().get(1).toString());
            break;

        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }

    if (sortOrder == null || sortOrder == ""){
        /**
         * By default sort on student names
         */
        sortOrder = NAME;
    }
    Cursor c = qb.query(db, projection, selection, selectionArgs,null, null,␣
↪sortOrder);

    /**
     * register to watch a content URI for changes
     */
    c.setNotificationUri(getContext().getContentResolver(), uri);
    return c;
}
```

While the user is providing a STUDENT_ID at content://sg.vp.owasp_mobile.provider.College/students, the query statement is prone to SQL injection. Obviously prepared statements must be used to avoid SQL injection, but input validation should also be applied so that only input that the app is expecting is processed.

All app functions that process data coming in through the UI should implement input validation:

- For user interface input, Android Saripaar v2 can be used.
- For input from IPC or URL schemes, a validation function should be created. For example, the following determines whether the string is alphanumeric:

```
public boolean isAlphaNumeric(String s){
    String pattern= "^[a-zA-Z0-9]*$";
    return s.matches(pattern);
}
```

An alternative to validation functions is type conversion, with, for example, Integer.parseInt if only integers are expected. The OWASP Input Validation Cheat Sheet contains more information about this topic.

Reference

- owasp-mastg Testing for Injection Flaws (MSTG-PLATFORM-2) Overview

Rulebook

- *Check deep links and verify correct relevance of web site (Required)*
- *Use IPC mechanism for security considerations (Required)*
- *All application functions that process data coming in from the UI should implement input validation (Required)*

6.2.3.1 Dynamic Analysis

The tester should manually test the input fields with strings like OR 1=1 – if, for example, a local SQL injection vulnerability has been identified.

On a rooted device, the command content can be used to query the data from a content provider. The following command queries the vulnerable function described above.

```
# content query --uri content://sg.vp.owasp_mobile.provider.College/students
```

SQL injection can be exploited with the following command. Instead of getting the record for Bob only, the user can retrieve all data.

```
# content query --uri content://sg.vp.owasp_mobile.provider.College/students --
↪where "name='Bob') OR 1=1--'"
```

Reference

- owasp-mastg Testing for Injection Flaws (MSTG-PLATFORM-2) Dynamic Analysis

6.2.4 Testing for Fragment Injection

Android SDK offers developers a way to present a Preferences activity to users, allowing the developers to extend and adapt this abstract class.

This abstract class parses the extra data fields of an Intent, in particular, the PreferenceActivity.EXTRA_SHOW_FRAGMENT(:android:show_fragment) and Preference Activity.EXTRA_SHOW_FRAGMENT_ARGUMENTS(:android:show_fragment_arguments) fields.

The first field is expected to contain the Fragment class name, and the second one is expected to contain the input bundle passed to the Fragment.

Because the PreferenceActivity uses reflection to load the fragment, an arbitrary class may be loaded inside the package or the Android SDK. The loaded class runs in the context of the application that exports this activity.

With this vulnerability, an attacker can call fragments inside the target application or run the code present in other classes' constructors. Any class that' s passed in the Intent and does not extend the Fragment class will cause a java.lang.CastException, but the empty constructor will be executed before the exception is thrown, allowing the code present in the class constructor run.

To prevent this vulnerability, a new method called isValidFragment was added in Android 4.4 (API level 19). It allows developers to override this method and define the fragments that may be used in this context.

The default implementation returns true on versions older than Android 4.4 (API level 19); it will throw an exception on later versions.

Reference

- owasp-mastg Testing for Injection Flaws (MSTG-PLATFORM-2) Testing for Fragment Injection (MSTG-PLATFORM-2)

6.2.4.1 Static Analysis

Steps:

- Check if android:targetSdkVersion less than 19.
- Find exported Activities that extend the PreferenceActivity class.
- Determine whether the method isValidFragment has been overridden.
- If the app currently sets its android:targetSdkVersion in the manifest to a value less than 19 and the vulnerable class does not contain any implementation of isValidFragment then, the vulnerability is inherited from the PreferenceActivity.
- In order to fix, developers should either update the android:targetSdkVersion to 19 or higher. Alternatively, if the android:targetSdkVersion cannot be updated, then developers should implement isValidFragment as described.

The following example shows an Activity that extends this activity:

```
public class MyPreferences extends PreferenceActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

The following examples show the isValidFragment method being overridden with an implementation that allows the loading of MyPreferenceFragment only:

```
@Override
protected boolean isValidFragment(String fragmentName)
{
return "com.fullpackage.MyPreferenceFragment".equals(fragmentName);
}
```

Reference

- owasp-mastg Testing for Fragment Injection (MSTG-PLATFORM-2) Static Analysis

6.2.4.2 Example of Vulnerable App and Exploitation

MainActivity.class

```
public class MainActivity extends PreferenceActivity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

MyFragment.class

```
public class MyFragment extends Fragment {
    public void onCreate (Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle␣
↪savedInstanceState) {
        View v = inflater.inflate(R.layout.fragmentLayout, null);
        WebView myWebView = (WebView) wv.findViewById(R.id.webview);
        myWebView.getSettings().setJavaScriptEnabled(true);
        myWebView.loadUrl(this.getActivity().getIntent().getDataString());
        return v;
    }
}
```

To exploit this vulnerable Activity, you can create an application with the following code:

```
Intent i = new Intent();
i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
i.setClassName("pt.claudio.insecurefragment","pt.claudio.insecurefragment.
↪MainActivity");
i.putExtra(":android:show_fragment","pt.claudio.insecurefragment.MyFragment");
i.setData(Uri.parse("https://security.claudio.pt"));
startActivity(i);
```

The Vulnerable App and Exploit PoC App are available for downloading.

Reference

- owasp-mastg Testing for Fragment Injection (MSTG-PLATFORM-2) Example of Vulnerable App and Exploitation

6.2.5 Testing for URL Loading in WebViews

WebViews are Android' s embedded components which allow your app to open web pages within your application. In addition to mobile apps related threats, WebViews may expose your app to common web threats (e.g. XSS, Open Redirect, etc.).

One of the most important things to do when testing WebViews is to make sure that only trusted content can be loaded in it. Any newly loaded page could be potentially malicious, try to exploit any WebView bindings or try to phish the user. Unless you' re developing a browser app, usually you' d like to restrict the pages being loaded to the domain of your app. A good practice is to prevent the user from even having the chance to input any URLs inside WebViews (which is the default on Android) nor navigate outside the trusted domains. Even when navigating on trusted domains there' s still the risk that the user might encounter and click on other links to untrustworthy content (e.g. if the page allows for other users to post comments). In addition, some developers might even override some default behavior which can be potentially dangerous for the user. See the Static Analysis section below for more details.

To provide a safer web browsing experience, Android 8.1 (API level 27) introduces the SafeBrowsing API, which allows your application to detect URLs that Google has classified as a known threat.

By default, WebViews show a warning to users about the security risk with the option to load the URL or stop the page from loading. With the SafeBrowsing API you can customize your application' s behavior by either reporting the threat to SafeBrowsing or performing a particular action such as returning back to safety each time it encounters a known threat. Please check the Android Developers documentation for usage examples.

You can use the SafeBrowsing API independently from WebViews using the SafetyNet library, which implements a client for Safe Browsing Network Protocol v4. SafetyNet allows you to analyze all the URLs that your app is supposed load. You can check URLs with different schemes (e.g. http, file) since SafeBrowsing is agnostic to URL schemes, and against TYPE_POTENTIALLY_HARMFUL_APPLICATION and TYPE_SOCIAL_ENGINEERING threat types.

Virus Total provides an API for analyzing URLs and local files for known threats. The API Reference is available on Virus Total developers page.

When sending URLs or files to be checked for known threats make sure they don't contain sensitive data which could compromise a user's privacy, or expose sensitive content from your application.

Reference

- owasp-mastg Testing for URL Loading in WebViews (MSTG-PLATFORM-2)

Rulebook

- *WebView can only load trusted content (Required)*

6.2.5.1 Static Analysis

As we mentioned before, handling page navigation should be analyzed carefully, especially when users might be able to navigate away from a trusted environment. The default and safest behavior on Android is to let the default web browser open any link that the user might click inside the WebView. However, this default logic can be modified by configuring a WebViewClient which allows navigation requests to be handled by the app itself. If this is the case, be sure to search for and inspect the following interception callback functions:

- shouldOverrideUrlLoading allows your application to either abort loading WebViews with suspicious content by returning true or allow the WebView to load the URL by returning false. Considerations:
  - This method is not called for POST requests.
  - This method is not called for XmlHttpRequests, iFrames, "src" attributes included in HTML or <script> tags. Instead, shouldInterceptRequest should take care of this.
- shouldInterceptRequest allows the application to return the data from resource requests. If the return value is null, the WebView will continue to load the resource as usual. Otherwise, the data returned by the shouldInterceptRequest method is used. Considerations:
  - This callback is invoked for a variety of URL schemes (e.g., http(s):, data:, file:, etc.), not only those schemes which send requests over the network.
  - This is not called for javascript: or blob: URLs, or for assets accessed via file:///android_asset/ or file:///android_res/ URLs. In the case of redirects, this is only called for the initial resource URL, not any subsequent redirect URLs.
- When Safe Browsing is enabled, these URLs still undergo Safe Browsing checks but the developer can allow the URL with setSafeBrowsingWhitelist or even ignore the warning via the onSafeBrowsingHit callback.

As you can see there are a lot of points to consider when testing the security of WebViews that have a WebViewClient configured, so be sure to carefully read and understand all of them by checking the WebViewClient Documentation.

While the default value of EnableSafeBrowsing is true, some applications might opt to disable it. To verify that SafeBrowsing is enabled, inspect the AndroidManifest.xml file and make sure that the configuration below is not present:

```
<manifest>
    <application>
        <meta-data android:name="android.webkit.WebView.EnableSafeBrowsing"
                   android:value="false" />
        ...
    </application>
</manifest>
```

Reference

- owasp-mastg Testing for Injection Flaws (MSTG-PLATFORM-2) Static Analysis

Rulebook

- *Ensure that all links that users may click within WebView open in their default web browser (Recommended)*

6.2.5.2 Dynamic Analysis

A convenient way to dynamically test deep linking is to use Frida or frida-trace and hook the shouldOverrideUrl-Loading, shouldInterceptRequest methods while using the app and clicking on links within the WebView. Be sure to also hook other related Uri methods such as getHost, getScheme or getPath which are typically used to inspect the requests and match known patterns or deny lists.

Reference

- owasp-mastg Testing for Injection Flaws (MSTG-PLATFORM-2) Dynamic Analysis

6.2.6 Rulebook

1. *Check WebView and investigate for unreliable input rendered by the app (Required)*
2. *Notes on input validation when using SharedPreferences (Required)*
3. *Notes on input validation when using public storage mechanisms other than SharedPreferences (Required)*
4. *All application functions that process data coming in from the UI should implement input validation (Required)*
5. *WebView can only load trusted content (Required)*
6. *Ensure that all links that users may click within WebView open in their default web browser (Recommended)*

6.2.6.1 Check WebView and investigate for unreliable input rendered by the app (Required)

WebView should be checked and investigated for untrusted input rendered by the app.

If some of the URLs opened by WebView are determined by user input, an XSS problem may exist.

The following sample code shows the XSS problem in Zoho Web Service as reported by Linus Särud as reported by Linus Särud.

Example in Java

```
webView.loadUrl("javascript:initialize(" + myNumber + ");");
```

Example in Kotlin

```
webView.loadUrl("javascript:initialize($myNumber);")
```

To avoid XSS problems, perform an input check on “myNumber” before using it in webView.loadUrl.

Also, the sample code below is a public overridden method of another example of an XSS problem determined by user input.

Example in Java

```
@Override
public boolean shouldOverrideUrlLoading(WebView view, String url) {
  if (url.substring(0,6).equalsIgnoreCase("yourscheme:")) {
    // parse the URL object and execute functions
  }
}
```

Example in Kotlin

```
    fun shouldOverrideUrlLoading(view: WebView, url: String): Boolean {
        if (url.substring(0, 6).equals("yourscheme:", ignoreCase = true)) {
            // parse the URL object and execute functions
        }
    }
```

To avoid XSS problems, perform a “url” input check before using “url” .

Verify that the following best practices have been followed:

- No untrusted data is rendered in HTML, JavaScript or other interpreted contexts unless it is absolutely necessary.
- Appropriate encoding is applied to escape characters, such as HTML entity encoding. Note: escaping rules become complicated when HTML is nested within other code, for example, rendering a URL located inside a JavaScript block.

Consider how data will be rendered in a response. For example, if data is rendered in a HTML context, six control characters that must be escaped:

Table 6.2.6.1.1 List of control characters that must be escaped

Character	Escaped
&	&amp;
<	<
>	>
“	"
‘	'
/	/

For a comprehensive list of escaping rules and other prevention measures, refer to the OWASP XSS Prevention Cheat Sheet.

If this is violated, the following may occur.

- A possible XSS problem exists.

6.2.6.2 Notes on input validation when using SharedPreferences (Required)

When using SharedPreferences as data storage, the following should be noted.

- In the case of String or StringSet, be careful how the data is interpreted. Sample code for using String in SharedPreferences

```
// Save setting value String
public static void saveString(Context ctx, String key, String val) {
    SharedPreferences prefs = ctx.getSharedPreferences(APP_NAME, Context.MODE_
↪PRIVATE);
    SharedPreferences.Editor editor = prefs.edit();
    editor.putString(key, val);
    editor.apply();
}

// Get setting value String
public static String loadString(Context ctx, String key) {
    SharedPreferences prefs = ctx.getSharedPreferences(APP_NAME, Context.MODE_
↪PRIVATE);
    return prefs.getString(key, "");
}
```

Sample code when using StringSet in SharedPreferences:.

```
// Save setting value Set<String>
public static void saveStringSet(Context ctx, String key, Set<String> vals) {
    SharedPreferences prefs = ctx.getSharedPreferences(APP_NAME, Context.MODE_
↪PRIVATE);
    SharedPreferences.Editor editor = prefs.edit();
    editor.putStringSet(key, vals);
    editor.apply();
```

(continues on next page)

(continued from previous page)

```
}

// Get setting value Set<String>
public static Set<String> loadStringSet(Context ctx, String key) {
    SharedPreferences prefs = ctx.getSharedPreferences(APP_NAME, Context.MODE_
↪PRIVATE);
    return prefs.getStringSet(key, new HashSet<String>());
}
```

- Check Android' s "Testing object persistence" to see how it should be verified.
- Use SharedPreferences.Editor to verify that you are storing and reading certificates and keys, considering vulnerabilities such as those found in Bouncy Castle and applying patches from security providers.

If this is violated, the following may occur.

- Strings are interpreted with the wrong data type.

6.2.6.3 Notes on input validation when using public storage mechanisms other than SharedPreferences (Required)

When using a public storage mechanism other than SharedPreferences, input must be validated at the time data is read from the storage mechanism.

The sample code below is an example of the process of reading a file from data storage.

```
private String readFile(String file){
    String text = null;
    try {
        FileInputStream fileInputStream = openFileInput(file);
        BufferedReader reader = new BufferedReader(new␣
↪InputStreamReader(fileInputStream, "UTF-8"));
        String lineBuffer;
        while (true){
            lineBuffer = reader.readLine();
            if (lineBuffer != null){
                text += lineBuffer;
            }
            else {
                break;
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return text;
}
```

When reading data safely, as in the sample code above, "text" should be verified before "text" is returned.

If this is violated, the following may occur.

- It may be read as unintended input.

6.2.6.4 All application functions that process data coming in from the UI should implement input validation (Required)

All application functions that process data coming in from the UI must implement input validation.

- Android Saripaar v2 can be used for user interface input.
  - Sample code omitted due to last release on 9/18/2015.
- For input from IPC or URL scheme, a validation function should be created.

  The sample code below is an example of a process to determine if a string is alphanumeric or not.

```
public boolean isAlphaNumeric(String s){
    String pattern= "^[a-zA-Z0-9]*$";
    return s.matches(pattern);
}
```

If this is violated, the following may occur.

- It may be read as unintended input.

6.2.6.5 WebView can only load trusted content (Required)

WebView is a built-in component of Android that allows web pages to be opened within an application. In addition to threats associated with mobile apps, WebView can also expose apps to common web threats (e.g., XSS, Open Redirect, etc.). Therefore, it is necessary to ensure that only trusted content can be loaded.

Safe Web Browsing.

For safer web browsing, use the SafeBrowsing API introduced in Android 8.1 (API level 27). This allows applications to detect URLs that Google has classified as known threats. By default, WebView displays an interstitial that alerts the user to known threats.

The sample code below is an example of how to instruct the WebView instance of an app to always return to a safe page when a known threat is detected.

The following is a declaration in AndroidManifest.

```
<manifest>
    <application>
        ...
        <meta-data android:name="android.webkit.WebView.EnableSafeBrowsing"
                   android:value="true" />
    </application>
</manifest>
```

The following is the callback process for the WebView calling class.

```
private var superSafeWebView: WebView? = null
private var safeBrowsingIsInitialized: Boolean = false

// ...

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    superSafeWebView = WebView(this).apply {
        webViewClient = MyWebViewClient()
        safeBrowsingIsInitialized = false
        startSafeBrowsing(this@SafeBrowsingActivity, { success ->
            safeBrowsingIsInitialized = true
            if (!success) {
                Log.e("MY_APP_TAG", "Unable to initialize Safe Browsing!")
            }
```

(continues on next page)

(continued from previous page)

```
        })
    }
}
```

The following is the processing in the callback onSafeBrowsingHit, which is called when an accessed URL is determined to be unsafe.

```
class MyWebViewClient : WebViewClient() {
    // Automatically go "back to safety" when attempting to load a website that
    // Safe Browsing has identified as a known threat. An instance of WebView
    // calls this method only after Safe Browsing is initialized, so there's no
    // conditional logic needed here.
    override fun onSafeBrowsingHit(
            view: WebView,
            request: WebResourceRequest,
            threatType: Int,
            callback: SafeBrowsingResponse
    ) {
        // The "true" argument indicates that your app reports incidents like
        // this one to Safe Browsing.
        callback.backToSafety(true)
        Toast.makeText(view.context, "Unsafe web page blocked.", Toast.LENGTH_
↪LONG).show()
    }
}
```

In addition, the SafetyNet library, which implements a client for the SafeBrowsing Network Protocol v4 SafeBrowsing API independently of WebViews. SafetyNet can analyze all URLs that an app plans to load. Since SafeBrowsing is URL scheme agnostic, it can check URLs with different schemes (http, file, etc.) to counter the TYPE_POTENTIALLY_HARMFUL_APPLICATION and TYPE_SOCIAL_ENGINEERING threat types. The following sample code is from the SafetyNet library.

The sample code below is an example of how to request a URL check using the SafetyNet library.

```
SafetyNet.getClient(this).lookupUri(
        url,
        SAFE_BROWSING_API_KEY,
        SafeBrowsingThreat.TYPE_POTENTIALLY_HARMFUL_APPLICATION,
        SafeBrowsingThreat.TYPE_SOCIAL_ENGINEERING
)
        .addOnSuccessListener(this) { sbResponse ->
            // Successful communication with service
            if (sbResponse.detectedThreats.isEmpty()) {
                // No threat found
            } else {
                // If a threat is found
            }
        }
        .addOnFailureListener(this) { e: Exception ->
            // Communication failure with service
            if (e is ApiException) {
                // If you get an error with the Google Play Services API
            } else {
                // If you get another error
            }
        }
```

Otherwise, Virus Total can be used to analyze URLs and local files for known threats.

Also, unless you are developing a browser app, you would normally limit the pages that are loaded to the app's domain. A good way to do this is to avoid giving users the opportunity to enter URLs within WebView (which is the default in Android) or even navigate outside of trusted domains. Even when navigating within a trusted domain,

there is still a risk that users will encounter links to untrusted content and click on them (for example, on pages where other users can post comments).

If this is violated, the following may occur.

- Vulnerable to web threats (e.g. XSS, Open Redirect, etc.).

6.2.6.6 Ensure that all links that users may click within WebView open in their default web browser (Recommended)

For Android to work by default and most securely, all links that a user might click on within WebView should open in the default web browser.

The following is a sample code that opens a link clicked in WebView in the default Web browser.

```
  @Override
  public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest␣
↪request) {
    Uri uri = request.getUri();
    String uriPath = uri.getPath();

    if(uriPathCheck(uriPath)){
      Intent intent = new Intent(Intent.ACTION_VIEW, uri)
      view.getContext().startActivity(intent);
      return true;
    }

    return false;
  }
```

However, this default logic can be changed by configuring WebViewClient to allow the app itself to handle navigation requests.

Therefore, consider the following considerations for the intercept callback function

- shouldOverrideUrlLoading allows the application to choose whether to return true to abort loading of a WebView with questionable content or false to allow the WebView to load the URL.

  Considerations :

  - This method is not called in a POST request.
  - This method is not called for src attributes in XmlHttpRequests, iFrames, HTML, or <scriptscreen>> tags. Instead, shouldInterceptRequest should handle this.

- shouldInterceptRequest allows the application to return data from a resource request. If the return value is NULL, WebView will continue loading the resource as usual. Otherwise, the data returned by the shouldInterceptRequest method is used.

  Considerations :

  - This callback is called for various URL schemes (http(s):, data:, file:, etc.) as well as schemes that send requests over the network.
  - It is not called for assets accessed via a javascript: or blob: URL, or a file:///android_asset/ or file:///android_res/ URL. In the case of redirects, this is only called for the first resource URL, not subsequent redirect URLs.
  - If Safe Browsing is enabled, these URLs will be subject to Safe Browsing checks, but the developer can allow the URLs with setSafeBrowsingWhitelist or ignore the warning with the onSafeBrowsingHit callback.

Safe Browsing can also be enabled by setting EnableSafeBrowsing to ture (default value). Note that setting the value to false disables Safe Browsing.

The following is an example of EnableSafeBrowsing configuration in AndroidManifest.xml.

```
<manifest>
    <application>
        <meta-data android:name="android.webkit.WebView.EnableSafeBrowsing"
                   android:value="true" />
        ...
    </application>
</manifest>
```

If this is violated, the following may occur.

- Unintended content may be loaded into your application.
- It is possible to run malicious JavaScript on your application.

6.3 MSTG-PLATFORM-3

The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected.

6.3.1 Deep Link Type

Deep links are URIs of any scheme that take users directly to specific content in an app. An app can set up deep links by adding intent filters on the Android Manifest and extracting data from incoming intents to navigate users to the correct activity.

Android supports two types of deep links:

- **Custom URL Schemes**, which are deep links that use any custom URL scheme, e.g. myapp:// (not verified by the OS).
- **Android App Links** (Android 6.0 (API level 23) and higher), which are deep links that use the http:// and https:// schemes and contain the autoVerify attribute (which triggers OS verification).

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Overview

6.3.2 Deep Link Collision

Using unverified deep links can cause a significant issue- any other apps installed on a user's device can declare and try to handle the same intent, which is known as deep link collision. Any arbitrary application can declare control over the exact same deep link belonging to another application.

In recent versions of Android this results in a so-called disambiguation dialog shown to the user that asks them to select the application that should handle the deep link. The user could make the mistake of choosing a malicious application instead of the legitimate one.

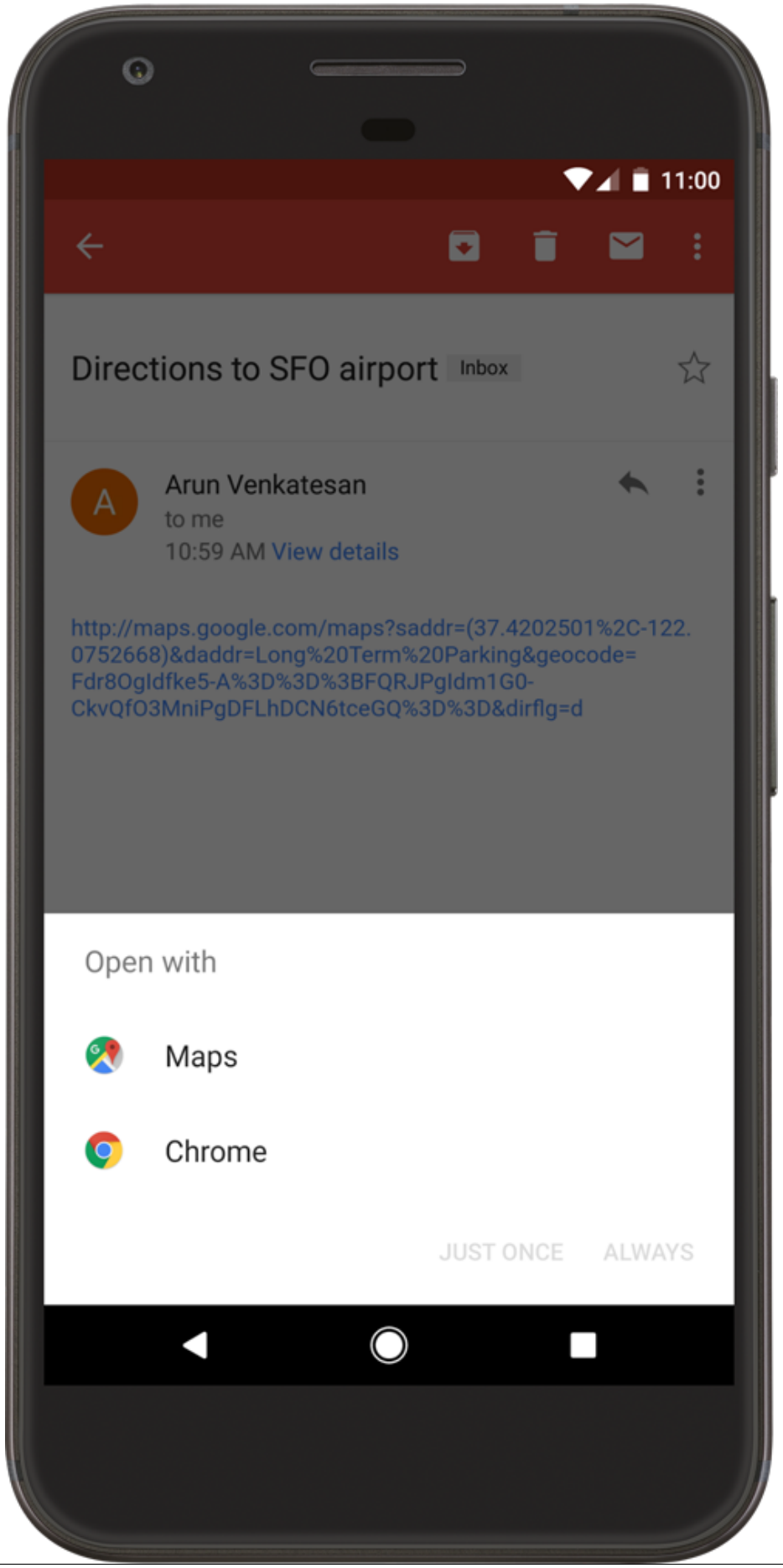


Fig 6.3.2.1 Disambiguation Dialog

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Deep Link Collision

6.3.3 Android App Links

In order to solve the deep link collision issue, Android 6.0 (API Level 23) introduced Android App Links, which are verified deep links based on a website URL explicitly registered by the developer. Clicking on an App Link will immediately open the app if it's installed.

There are some key differences from unverified deep links:

- App Links only use http:// and https:// schemes, any other custom URL schemes are not allowed.
- App Links require a live domain to serve a Digital Asset Links file via HTTPS.
- App Links do not suffer from deep link collision since they don't show a disambiguation dialog when a user opens them.

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Android App Links

6.3.4 Testing Deep Links

Any existing deep links (including App Links) can potentially increase the app attack surface. This includes many risks such as link hijacking, sensitive functionality exposure, etc. The Android version in which the app runs also influences the risk:

- Before Android 12 (API level 31), if the app has any non-verifiable links, it can cause the system to not verify all Android App Links for that app.
- Starting on Android 12 (API level 31), apps benefit from a reduced attack surface. A generic web intent resolves to the user's default browser app unless the target app is approved for the specific domain contained in that web intent.

All deep links must be enumerated and verified for correct website association. The actions they perform must be well tested, especially all input data, which should be deemed untrustworthy and thus should always be validated.

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Testing Deep Links

Rulebook

- *Check deep links and verify correct relevance of web site (Required)*

6.3.5 Static Analysis

6.3.5.1 Enumerate Deep Links

Inspecting the Android Manifest:

You can easily determine whether deep links (with or without custom URL schemes) are defined by decoding the app using apktool and inspecting the Android Manifest file looking for <intent-filter> elements.

- **Custom Url Schemes**: The following example specifies a deep link with a custom URL scheme called myapp://.

```
<activity android:name=".MyUriActivity">
  <intent-filter>
      <action android:name="android.intent.action.VIEW" />
      <category android:name="android.intent.category.DEFAULT" />
```

(continues on next page)

(continued from previous page)

```
      <category android:name="android.intent.category.BROWSABLE" />
      <data android:scheme="myapp" android:host="path" />
  </intent-filter>
</activity>
```

- **Deep Links**: The following example specifies a deep Link using both the http:// and https:// schemes, along with the host and path that will activate it (in this case, the full URL would be https://www.myapp.com/my/app/path):

```
<intent-filter>
  ...
  <data android:scheme="http" android:host="www.myapp.com" android:path="/my/app/
↪path" />
  <data android:scheme="https" android:host="www.myapp.com" android:path="/my/app/
↪path" />
</intent-filter>
```

- **App Links**: If the <intent-filter> includes the flag android:autoVerify=" true" , this causes the Android system to reach out to the declared android:host in an attempt to access the Digital Asset Links file in order to verify the App Links. A deep link can be considered an App Link only if the verification is successful.

```
<intent-filter android:autoVerify="true">
```

When listing deep links remember that <data> elements within the same <intent-filter> are actually merged together to account for all variations of their combined attributes.

```
<intent-filter>
  ...
  <data android:scheme="https" android:host="www.example.com" />
  <data android:scheme="app" android:host="open.my.app" />
</intent-filter>
```

It might seem as though this supports only https://www.example.com and app://open.my.app. However, it actually supports:

- https://www.example.com
- app://open.my.app
- app://www.example.com
- https://open.my.app

Using Dumpsys:

Use adb to run the following command that will show all schemes:

```
adb shell dumpsys package com.example.package
```

Using Android "App Link Verification" Tester:

Use the Android "App Link Verification" Tester script to list all deep links (list-all) or only app links (list-applinks):

```
python3 deeplink_analyser.py -op list-all -apk ~/Downloads/example.apk

.MainActivity

app://open.my.app
app://www.example.com
https://open.my.app
https://www.example.com
```

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Enumerate Deep Links

Rulebook

- *Check deep links and verify correct relevance of web site (Required)*

6.3.5.2 Check for Correct Website Association

Even if deep links contain the android:autoVerify=" true" attribute, they must be actually verified in order to be considered App Links. You should test for any possible misconfigurations that might prevent full verification.

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Check for Correct Website Association

Rulebook

- *Check deep links and verify correct relevance of web site (Required)*

Automatic Verification

Use the Android "App Link Verification" Tester script to get the verification status for all app links (verify-applinks). See an example here.

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Static Analysis Automatic Verification

Only on Android 12 (API level 31) or higher:

You can use adb to test the verification logic regardless of whether the app targets Android 12 (API level 31) or not. This feature allows you to:

- invoke the verification process manually.
- reset the state of the target app' s Android App Links on your device.
- invoke the domain verification process.

You can also review the verification results. For example:

```
adb shell pm get-app-links com.example.package

com.example.package:
    ID: 01234567-89ab-cdef-0123-456789abcdef
    Signatures: [***]
    Domain verification state:
      example.com: verified
      sub.example.com: legacy_failure
      example.net: verified
      example.org: 1026
```

The same information can be found by running adb shell dumpsys package com.example.package (only on Android 12 (API level 31) or higher).

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Automatic Verification

Manual Verification

This section details a few, of potentially many, reasons why the verification process failed or was not actually triggered. See more information in the Android Developers Documentation and in the white paper "Measuring the Insecurity of Mobile Deep Links of Android" .

Check the Digital Asset Links file:

- Check for missing Digital Asset Links file:

  - try to find it in the domain's /.well-known/ path. Example: https://www.example.com/.well-known/assetlinks.json
  - or try https://digitalassetlinks.googleapis.com/v1/statements:list?source.web.site=www.example.com
- Check for valid Digital Asset Links file **served via HTTP**.
- Check for invalid Digital Asset Links files served via HTTPS. For example:
  - the file contains invalid JSON.
  - the file doesn't include the target app's package.

Check for Redirects:

To enhance the app security, the system doesn't verify any Android App Links for an app if the server sets a redirect such as http://example.com to https://example.com or example.com to www.example.com.

Check for Subdomains:

If an intent filter lists multiple hosts with different subdomains, there must be a valid Digital Asset Links file on each domain. For example, the following intent filter includes www.example.com and mobile.example.com as accepted intent URL hosts.

```
<application>
  <activity android:name="MainActivity">
    <intent-filter android:autoVerify="true">
      <action android:name="android.intent.action.VIEW" />
      <category android:name="android.intent.category.DEFAULT" />
      <category android:name="android.intent.category.BROWSABLE" />
      <data android:scheme="https" />
      <data android:scheme="https" />
      <data android:host="www.example.com" />
      <data android:host="mobile.example.com" />
    </intent-filter>
  </activity>
</application>
```

In order for the deep links to correctly register, a valid Digital Asset Links file must be published at both https://www.example.com/.well-known/assetlinks.json and https://mobile.example.com/.well-known/assetlinks.json.

Check for Wildcards:

If the hostname includes a wildcard (such as *.example.com), you should be able to find a valid Digital Asset Links file at the root hostname: https://example.com/.well-known/assetlinks.json.

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Manual Verification

6.3.5.3 Check the Handler Method

Even if the deep link is correctly verified, the logic of the handler method should be carefully analyzed. Pay special attention to deep links being used to transmit data (which is controlled externally by the user or any other app).

First, obtain the name of the Activity from the Android Manifest <activity> element which defines the target <intent-filter> and search for usage of getIntent and getData. This general approach of locating these methods can be used across most applications when performing reverse engineering and is key when trying to understand how the application uses deep links and handles any externally provided input data and if it could be subject to any kind of abuse.

The following example is a snippet from an exemplary Kotlin app decompiled with jadx. From the static analysis we know that it supports the deep link deeplinkdemo://load.html/ as part of com.mstg.deeplinkdemo.WebViewActivity.

```
// snippet edited for simplicity
public final class WebViewActivity extends AppCompatActivity {
    private ActivityWebViewBinding binding;

    public void onCreate(Bundle savedInstanceState) {
        Uri data = getIntent().getData();
        String html = data == null ? null : data.getQueryParameter("html");
        Uri data2 = getIntent().getData();
        String deeplink_url = data2 == null ? null : data2.getQueryParameter("url
↪");
        View findViewById = findViewById(R.id.webView);
        if (findViewById != null) {
            WebView wv = (WebView) findViewById;
            wv.getSettings().setJavaScriptEnabled(true);
            if (deeplink_url != null) {
                wv.loadUrl(deeplink_url);
            ...
```

You can simply follow the deeplink_url String variable and see the result from the wv.loadUrl call. This means the attacker has full control of the URL being loaded to the WebView (as shown above has JavaScript enabled.

The same WebView might be also rendering an attacker controlled parameter. In that case, the following deep link payload would trigger Reflected Cross-Site Scripting (XSS) within the context of the WebView:

```
deeplinkdemo://load.html?attacker_controlled=<svg onload=alert(1)>
```

But there are many other possibilities. Be sure to check the following sections to learn more about what to expect and how to test different scenarios:

- "Cross-Site Scripting Flaws (MSTG-PLATFORM-2)" .
- "Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2)" .
- "Testing Object Persistence (MSTG-PLATFORM-8)" .
- "Testing for URL Loading in WebViews (MSTG-PLATFORM-2)"
- "Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5)"
- "Testing WebView Protocol Handlers (MSTG-PLATFORM-6)"

In addition, we recommend to search and read public reports (search term: "deep link*" |" deeplink*" site:https://hackerone.com/reports/). For example:

- "[HackerOne#1372667] Able to steal bearer token from deep link"
- "[HackerOne#401793] Insecure deeplink leads to sensitive information disclosure"
- "[HackerOne#583987] Android app deeplink leads to CSRF in follow action"
- "[HackerOne#637194] Bypass of biometrics security functionality is possible in Android application"
- "[HackerOne#341908] XSS via Direct Message deeplinks"

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Check the Handler Method

Rulebook

- *Carefully analyze the logic of the Handler Method even if deep links are correctly validated (Required)*

6.3.6 Dynamic Analysis

Here you will use the list of deep links from the static analysis to iterate and determine each handler method and the processed data, if any. You will first start a Frida hook and then begin invoking the deep links.

The following example assumes a target app that accepts this deep link: deeplinkdemo://load.html. However, we don't know the corresponding handler method yet, nor the parameters it potentially accepts.

[Step 1] Frida Hooking:

You can use the script "Android Deep Link Observer" from Frida CodeShare to monitor all invoked deep links triggering a call to Intent.getData. You can also use the script as a base to include your own modifications depending on the use case at hand. In this case we included the stack trace in the script since we are interested in the method which calls Intent.getData.

[Step 2] Invoking Deep Links:

Now you can invoke any of the deep links using adb and the Activity Manager (am) which will send intents within the Android device. For example:

```
adb shell am start -W -a android.intent.action.VIEW -d "deeplinkdemo://load.html/?
↪message=ok#part1"

Starting: Intent { act=android.intent.action.VIEW dat=deeplinkdemo://load.html/?
↪message=ok }
Status: ok
LaunchState: WARM
Activity: com.mstg.deeplinkdemo/.WebViewActivity
TotalTime: 210
WaitTime: 217
Complete
```

This might trigger the disambiguation dialog when using the "http/https" schema or if other installed apps support the same custom URL schema. You can include the package name to make it an explicit intent.

This invocation will log the following:

```
[*] Intent.getData() was called
[*] Activity: com.mstg.deeplinkdemo.WebViewActivity
[*] Action: android.intent.action.VIEW

[*] Data
- Scheme: deeplinkdemo://
- Host: /load.html
- Params: message=ok
- Fragment: part1

[*] Stacktrace:

android.content.Intent.getData(Intent.java)
com.mstg.deeplinkdemo.WebViewActivity.onCreate(WebViewActivity.kt)
android.app.Activity.performCreate(Activity.java)
...
com.android.internal.os.ZygoteInit.main(ZygoteInit.java)
```

In this case we've crafted the deep link including arbitrary parameters (?message=ok) and fragment (#part1). We still don't know if they are being used. The information above reveals useful information that you can use now to reverse engineer the app. See the section "Check the Handler Method" to learn about things you should consider.

- File: WebViewActivity.kt
- Class: com.mstg.deeplinkdemo.WebViewActivity
- Method: onCreate

Sometimes you can even take advantage of other applications that you know interact with your target app. You can reverse engineer the app, (e.g. to extract all strings and filter those which include the target deep links, deeplinkdemo:///load.html in the previous case), or use them as triggers, while hooking the app as previously discussed.

Reference

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Dynamic Analysis

6.3.7 Rulebook

1. *Check deep links and verify correct relevance of web site (Required)*
2. *Carefully analyze the logic of the Handler Method even if deep links are correctly validated (Required)*

6.3.7.1 Check deep links and verify correct relevance of web site (Required)

Existing deep links (including app links) may expand the app' s attack surface area, which includes link hijacking, leakage of sensitive functionality, and many other risks.

To prevent the above, all deep links should be enumerated and the correct relevance of the website should be verified. In particular, input data is considered unreliable and must be constantly verified.

Deep link enumeration.

You can easily determine whether deep links (with or without custom URL schemes) are defined by decoding the app using apktool and inspecting the Android Manifest file looking for <intent-filter> elements.

- **Custom Url Schemes**: The following example specifies a deep link with a custom URL scheme called myapp://.

```
<activity android:name=".MyUriActivity">
  <intent-filter>
      <action android:name="android.intent.action.VIEW" />
      <category android:name="android.intent.category.DEFAULT" />
      <category android:name="android.intent.category.BROWSABLE" />
      <data android:scheme="myapp" android:host="path" />
  </intent-filter>
</activity>
```

- **Deep Links**: The following example specifies a deep Link using both the http:// and https:// schemes, along with the host and path that will activate it (in this case, the full URL would be https://www.myapp.com/my/app/path):

```
<intent-filter>
  ...
  <data android:scheme="http" android:host="www.myapp.com" android:path="/my/app/
↪path" />
  <data android:scheme="https" android:host="www.myapp.com" android:path="/my/app/
↪path" />
</intent-filter>
```

- **App Links**: If the <intent-filter> includes the flag android:autoVerify=" true" , this causes the Android system to reach out to the declared android:host in an attempt to access the Digital Asset Links file in order to verify the App Links. A deep link can be considered an App Link only if the verification is successful.

```
<intent-filter android:autoVerify="true">
```

When listing deep links remember that <data> elements within the same <intent-filter> are actually merged together to account for all variations of their combined attributes.

```
<intent-filter>
  ...
  <data android:scheme="https" android:host="www.example.com" />
  <data android:scheme="app" android:host="open.my.app" />
</intent-filter>
```

It might seem as though this supports only https://www.example.com and app://open.my.app. However, it actually supports:

- https://www.example.com
- app://open.my.app
- app://www.example.com
- https://open.my.app

Even if the deep link includes the android:autoVerify=" true" attribute, it must actually be verified in order to be considered an app link. It should be tested for any misconfigurations that might prevent full validation.

6.3.7.2 Carefully analyze the logic of the Handler Method even if deep links are correctly validated (Required)

Even if deep linking is correctly verified, the logic of the Handler Method must be carefully analyzed. In particular, care should be taken when deeplinks are used to transmit data (controlled externally by the user or other applications).

Note that in the sample code below, the results of the wv.loadUrl call can be seen by simply traversing the deeplink_url string variable.

```
// snippet edited for simplicity
public final class WebViewActivity extends AppCompatActivity {
    private ActivityWebViewBinding binding;

    public void onCreate(Bundle savedInstanceState) {
        Uri data = getIntent().getData();
        String html = data == null ? null : data.getQueryParameter("html");
        Uri data2 = getIntent().getData();
        String deeplink_url = data2 == null ? null : data2.getQueryParameter("url
↪");
        View findViewById = findViewById(R.id.webView);
        if (findViewById != null) {
            WebView wv = (WebView) findViewById;
            wv.getSettings().setJavaScriptEnabled(true);
            if (deeplink_url != null) {
                wv.loadUrl(deeplink_url);
            ...
```

The same WebView might be also rendering an attacker controlled parameter. In that case, the following deep link payload would trigger Reflected Cross-Site Scripting (XSS) within the context of the WebView:

```
deeplinkdemo://load.html?attacker_controlled=<svg onload=alert(1)>
```

But there are many other possibilities. Be sure to check the following sections to learn more about what to expect and how to test different scenarios:

- "Cross-Site Scripting Flaws (MSTG-PLATFORM-2)" .
- "Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2)" .
- "Testing Object Persistence (MSTG-PLATFORM-8)" .
- "Testing for URL Loading in WebViews (MSTG-PLATFORM-2)"
- "Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5)"
- "Testing WebView Protocol Handlers (MSTG-PLATFORM-6)"

In addition, we recommend to search and read public reports (search term: "deep link*" |" deeplink*" site:https://hackerone.com/reports/). For example:

- "[HackerOne#1372667] Able to steal bearer token from deep link"
- "[HackerOne#401793] Insecure deeplink leads to sensitive information disclosure"
- "[HackerOne#583987] Android app deeplink leads to CSRF in follow action"
- "[HackerOne#637194] Bypass of biometrics security functionality is possible in Android application"
- "[HackerOne#341908] XSS via Direct Message deeplinks"

6.4 MSTG-PLATFORM-4

The app does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected.

6.4.1 Disclosure of confidential functions by IPC

During implementation of a mobile application, developers may apply traditional techniques for IPC (such as using shared files or network sockets). The IPC system functionality offered by mobile application platforms should be used because it is much more mature than traditional techniques. Using IPC mechanisms with no security in mind may cause the application to leak or expose sensitive data.

The following is a list of Android IPC Mechanisms that may expose sensitive data:

- Binders
- Services
- Bound Services
- AIDL
- Intents
- Content Providers

Reference

- owasp-mastgTesting for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Overview

Rulebook

- *Use IPC mechanism for security considerations (Required)*

6.4.1.1 Activities

Inspect the AndroidManifest

In the "Sieve" app, we find three exported activities, identified by <activity>:

```
<activity android:excludeFromRecents="true" android:label="@string/app_name"␣
↪android:launchMode="singleTask" android:name=".MainLoginActivity"␣
↪android:windowSoftInputMode="adjustResize|stateVisible">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity android:clearTaskOnLaunch="true" android:excludeFromRecents="true"␣
↪android:exported="true" android:finishOnTaskLaunch="true" android:label="@string/
↪title_activity_file_select" android:name=".FileSelectActivity" />
<activity android:clearTaskOnLaunch="true" android:excludeFromRecents="true"␣
```

(continues on next page)

(continued from previous page)

```
↪android:exported="true" android:finishOnTaskLaunch="true" android:label="@string/
↪title_activity_pwlist" android:name=".PWList" />
```

Inspect the Source Code

By inspecting the PWList.java activity, we see that it offers options to list all keys, add, delete, etc. If we invoke it directly, we will be able to bypass the LoginActivity. More on this can be found in the dynamic analysis below.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Activities

Rulebook

- *Use IPC mechanism for security considerations (Required)*

6.4.1.2 Services

Inspect the AndroidManifest

In the "Sieve" app, we find two exported services, identified by <service>:

```
<service android:exported="true" android:name=".AuthService" android:process=
↪":remote" />
<service android:exported="true" android:name=".CryptoService" android:process=
↪":remote" />
```

Inspect the Source Code

Check the source code for the class android.app.Service:

By reversing the target application, we can see that the service AuthService provides functionality for changing the password and PIN-protecting the target app.

```
   public void handleMessage(Message msg) {
            AuthService.this.responseHandler = msg.replyTo;
            Bundle returnBundle = msg.obj;
            int responseCode;
            int returnVal;
            switch (msg.what) {
                ...
                case AuthService.MSG_SET /*6345*/:
                    if (msg.arg1 == AuthService.TYPE_KEY) /*7452*/ {
                        responseCode = 42;
                        if (AuthService.this.setKey(returnBundle.getString("com.
↪mwr.example.sieve.PASSWORD"))) {
                            returnVal = 0;
                        } else {
                            returnVal = 1;
                        }
                    } else if (msg.arg1 == AuthService.TYPE_PIN) {
                        responseCode = 41;
                        if (AuthService.this.setPin(returnBundle.getString("com.
↪mwr.example.sieve.PIN"))) {
                            returnVal = 0;
                        } else {
                            returnVal = 1;
                        }
                    } else {
                        sendUnrecognisedMessage();
                        return;
                    }
            }
   }
```

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Services

Rulebook

- *Use IPC mechanism for security considerations (Required)*

6.4.1.3 Broadcast Receivers

Inspect the AndroidManifest

In the "Android Insecure Bank" app, we find a broadcast receiver in the manifest, identified by <receiver>:

```
<receiver android:exported="true" android:name="com.android.insecurebankv2.
↪MyBroadCastReceiver">
    <intent-filter>
        <action android:name="theBroadcast" />
    </intent-filter>
</receiver>
```

Inspect the Source Code

Search the source code for strings like sendBroadcast, sendOrderedBroadcast, and sendStickyBroadcast. Make sure that the application doesn't send any sensitive data.

If an Intent is broadcasted and received within the application only, LocalBroadcastManager can be used to prevent other apps from receiving the broadcast message. This reduces the risk of leaking sensitive information.

To understand more about what the receiver is intended to do, we have to go deeper in our static analysis and search for usage of the class android.content.BroadcastReceiver and the Context.registerReceiver method, which is used to dynamically create receivers.

The following extract of the target application's source code shows that the broadcast receiver triggers transmission of an SMS message containing the user's decrypted password.

```
public class MyBroadCastReceiver extends BroadcastReceiver {
  String usernameBase64ByteString;
  public static final String MYPREFS = "mySharedPreferences";

  @Override
  public void onReceive(Context context, Intent intent) {
    // TODO Auto-generated method stub

        String phn = intent.getStringExtra("phonenumber");
        String newpass = intent.getStringExtra("newpass");

    if (phn != null) {
      try {
                SharedPreferences settings = context.getSharedPreferences(MYPREFS,␣
↪Context.MODE_WORLD_READABLE);
                final String username = settings.getString("EncryptedUsername",␣
↪null);
                byte[] usernameBase64Byte = Base64.decode(username, Base64.
↪DEFAULT);
                usernameBase64ByteString = new String(usernameBase64Byte, "UTF-8");
                final String password = settings.getString("superSecurePassword",␣
↪null);
                CryptoClass crypt = new CryptoClass();
                String decryptedPassword = crypt.aesDeccryptedString(password);
                String textPhoneno = phn.toString();
                String textMessage = "Updated Password from: "+decryptedPassword+"␣
↪to: "+newpass;
                SmsManager smsManager = SmsManager.getDefault();
```

(continues on next page)

(continued from previous page)

```
                System.out.println("For the changepassword - phonenumber:
↪"+textPhoneno+" password is: "+textMessage);
smsManager.sendTextMessage(textPhoneno, null, textMessage, null, null);
            }
        }
    }
}
```

BroadcastReceivers should use the android:permission attribute; otherwise, other applications can invoke them. You can use Context.sendBroadcast(intent, receiverPermission); to specify permissions a receiver must have to read the broadcast. You can also set an explicit application package name that limits the components this Intent will resolve to. If left as the default value (null), all components in all applications will be considered. If non-null, the Intent can match only the components in the given application package.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Broadcast Receivers

Rulebook

- *Use IPC mechanism for security considerations (Required)*
- *Note that authorization may be required on both the receiving and sending sides of Broadcast (Required)*

6.4.2 Static Analysis

We start by looking at the AndroidManifest.xml, where all activities, services, and content providers included in the source code must be declared (otherwise the system won't recognize them and they won't run). Broadcast receivers can be declared in the manifest or created dynamically. You will want to identify elements such as

- <intent-filter>
- <service>
- <provider>
- <receiver>

An "exported" activity, service, or content can be accessed by other apps. There are two common ways to designate a component as exported. The obvious one is setting the export tag to true android:exported="true". The second way involves defining an <intent-filter> within the component element (<activity>, <service>, <receiver>). When this is done, the export tag is automatically set to "true". To prevent all other Android apps from interacting with the IPC component element, be sure that the android:exported="true" value and an <intent-filter> aren't in their AndroidManifest.xml files unless this is necessary.

Remember that using the permission tag (android:permission) will also limit other applications' access to a component. If your IPC is intended to be accessible to other applications, you can apply a security policy with the <permission> element and set a proper android:protectionLevel. When android:permission is used in a service declaration, other applications must declare a corresponding <uses-permission> element in their own manifest to start, stop, or bind to the service.

For more information about the content providers, please refer to the test case "Testing Whether Stored Sensitive Data Is Exposed via IPC Mechanisms" in chapter "Testing Data Storage".

Once you identify a list of IPC mechanisms, review the source code to see whether sensitive data is leaked when the mechanisms are used. For example, content providers can be used to access database information, and services can be probed to see if they return data. Broadcast receivers can leak sensitive information if probed or sniffed.

In the following, we use two example apps and give examples of identifying vulnerable IPC components:

- "Sieve"

  Reference

  - *Activities*
  - *Services*
- "Android Insecure Bank"

  Reference

  - *Broadcast Receivers*

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Static Analysis

Rulebook

- *Use IPC mechanism for security considerations (Required)*

6.4.3 Dynamic Analysis

You can enumerate IPC components with MobSF. To list all exported IPC components, upload the APK file and the components collection will be displayed in the following screen:

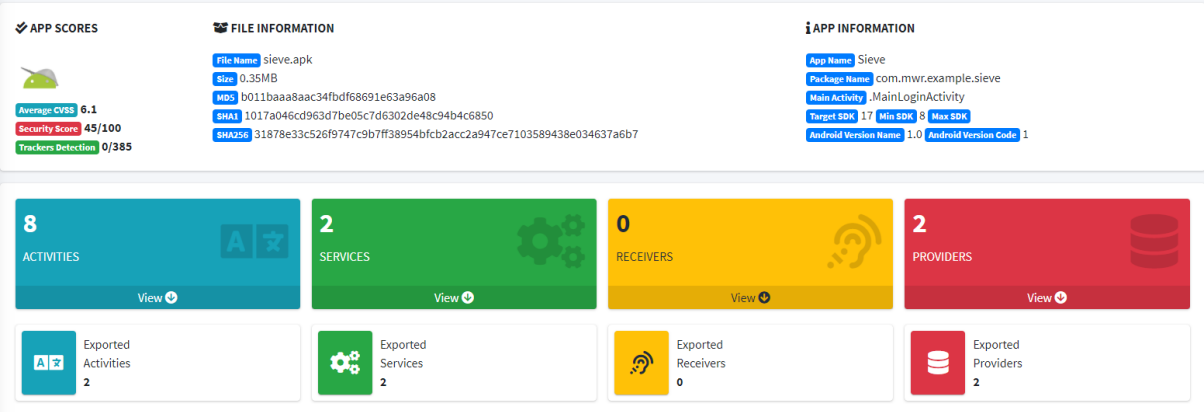


Fig 6.4.3.1 Component Collection

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Dynamic Analysis

6.4.3.1 Content Providers

The "Sieve" application implements a vulnerable content provider. To list the content providers exported by the Sieve app, execute the following command:

```
$ adb shell dumpsys package com.mwr.example.sieve | grep -Po "Provider{[\w\d\s\./
↪]+}" | sort -u
Provider{34a20d5 com.mwr.example.sieve/.FileBackupProvider}
Provider{64f10ea com.mwr.example.sieve/.DBContentProvider}
```

Once identified, you can use jadx to reverse engineer the app and analyze the source code of the exported content providers to identify potential vulnerabilities.

To identify the corresponding class of a content provider, use the following information:

- Package Name: com.mwr.example.sieve.
- Content Provider Class Name: DBContentProvider.

When analyzing the class com.mwr.example.sieve.DBContentProvider, you'll see that it contains several URIs:

```
package com.mwr.example.sieve;
...
public class DBContentProvider extends ContentProvider {
    public static final Uri KEYS_URI = Uri.parse("content://com.mwr.example.sieve.
↪DBContentProvider/Keys");
    public static final Uri PASSWORDS_URI = Uri.parse("content://com.mwr.example.
↪sieve.DBContentProvider/Passwords");
...
}
```

Use the following commands to call the content provider using the identified URIs:

```
$ adb shell content query --uri content://com.mwr.example.sieve.DBContentProvider/
↪Keys/
Row: 0 Password=1234567890AZERTYUIOPazertyuiop, pin=1234

$ adb shell content query --uri content://com.mwr.example.sieve.DBContentProvider/
↪Passwords/
Row: 0 _id=1, service=test, username=test, password=BLOB, email=t@tedt.com
Row: 1 _id=2, service=bank, username=owasp, password=BLOB, email=user@tedt.com

$ adb shell content query --uri content://com.mwr.example.sieve.DBContentProvider/
↪Passwords/ --projection email:username:password --where 'service=\"bank\"'
Row: 0 email=user@tedt.com, username=owasp, password=BLOB
```

You are able now to retrieve all database entries (see all lines starting with "Row:" in the output).

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Content Providers

6.4.3.2 Activity

To list activities exported by an application, you can use the following command and focus on activity elements:

```
$ aapt d xmltree sieve.apk AndroidManifest.xml
...
E: activity (line=32)
  A: android:label(0x01010001)=@0x7f05000f
  A: android:name(0x01010003)=".FileSelectActivity" (Raw: ".FileSelectActivity")
  A: android:exported(0x01010010)=(type 0x12)0xffffffff
  A: android:finishOnTaskLaunch(0x01010014)=(type 0x12)0xffffffff
  A: android:clearTaskOnLaunch(0x01010015)=(type 0x12)0xffffffff
  A: android:excludeFromRecents(0x01010017)=(type 0x12)0xffffffff
E: activity (line=40)
  A: android:label(0x01010001)=@0x7f050000
  A: android:name(0x01010003)=".MainLoginActivity" (Raw: ".MainLoginActivity")
  A: android:excludeFromRecents(0x01010017)=(type 0x12)0xffffffff
  A: android:launchMode(0x0101001d)=(type 0x10)0x2
  A: android:windowSoftInputMode(0x0101022b)=(type 0x11)0x14
  E: intent-filter (line=46)
    E: action (line=47)
      A: android:name(0x01010003)="android.intent.action.MAIN" (Raw: "android.
↪intent.action.MAIN")
    E: category (line=49)
      A: android:name(0x01010003)="android.intent.category.LAUNCHER" (Raw:
↪"android.intent.category.LAUNCHER")
E: activity (line=52)
  A: android:label(0x01010001)=@0x7f050009
  A: android:name(0x01010003)=".PWList" (Raw: ".PWList")
```

(continues on next page)

(continued from previous page)

```
  A: android:exported(0x01010010)=(type 0x12)0xffffffff
  A: android:finishOnTaskLaunch(0x01010014)=(type 0x12)0xffffffff
  A: android:clearTaskOnLaunch(0x01010015)=(type 0x12)0xffffffff
  A: android:excludeFromRecents(0x01010017)=(type 0x12)0xffffffff
E: activity (line=60)
  A: android:label(0x01010001)=@0x7f05000a
  A: android:name(0x01010003)=".SettingsActivity" (Raw: ".SettingsActivity")
  A: android:finishOnTaskLaunch(0x01010014)=(type 0x12)0xffffffff
  A: android:clearTaskOnLaunch(0x01010015)=(type 0x12)0xffffffff
  A: android:excludeFromRecents(0x01010017)=(type 0x12)0xffffffff
...
```

You can identify an exported activity using one of the following properties:

- It have an intent-filter sub declaration.
- It have the attribute android:exported to 0xffffffff.

You can also use jadx to identify exported activities in the file AndroidManifest.xml using the criteria described above:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.
↪mwr.example.sieve">
...
  <!-- This activity is exported via the attribute "exported" -->
  <activity android:name=".FileSelectActivity" android:exported="true" />
   <!-- This activity is exported via the "intent-filter" declaration  -->
  <activity android:name=".MainLoginActivity">
    <intent-filter>
      <action android:name="android.intent.action.MAIN"/>
      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
  </activity>
  <!-- This activity is exported via the attribute "exported" -->
  <activity android:name=".PWList" android:exported="true" />
  <!-- Activities below are not exported -->
  <activity android:name=".SettingsActivity" />
  <activity android:name=".AddEntryActivity"/>
  <activity android:name=".ShortLoginActivity" />
  <activity android:name=".WelcomeActivity" />
  <activity android:name=".PINActivity" />
...
</manifest>
```

Enumerating activities in the vulnerable password manager "Sieve" shows that the following activities are exported:

- .MainLoginActivity
- .PWList
- .FileSelectActivity

Use the command below to launch an activity:

```
# Start the activity without specifying an action or an category
$ adb shell am start -n com.mwr.example.sieve/.PWList
Starting: Intent { cmp=com.mwr.example.sieve/.PWList }

# Start the activity indicating an action (-a) and an category (-c)
$ adb shell am start -n "com.mwr.example.sieve/.MainLoginActivity" -a android.
↪intent.action.MAIN -c android.intent.category.LAUNCHER
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.
↪LAUNCHER] cmp=com.mwr.example.sieve/.MainLoginActivity }
```

Since the activity .PWList is called directly in this example, you can use it to bypass the login form protecting the password manager, and access the data contained within the password manager.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Activities

6.4.3.3 Service

Services can be enumerated with the Drozer module app.service.info:

```
dz> run app.service.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
  com.mwr.example.sieve.AuthService
    Permission: null
  com.mwr.example.sieve.CryptoService
    Permission: null
```

To communicate with a service, you must first use static analysis to identify the required inputs.

Because this service is exported, you can use the module app.service.send to communicate with the service and change the password stored in the target application:

```
dz> run app.service.send com.mwr.example.sieve com.mwr.example.sieve.AuthService --
↪msg 6345 7452 1 --extra string com.mwr.example.sieve.PASSWORD "abcdabcdabcdabcd"␣
↪--bundle-as-obj
Got a reply from com.mwr.example.sieve/com.mwr.example.sieve.AuthService:
  what: 4
  arg1: 42
  arg2: 0
  Empty
```

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Services

6.4.3.4 BroadcastReceivers

To list broadcast receivers exported by an application, you can use the following command and focus on receiver elements:

```
$ aapt d xmltree InsecureBankv2.apk AndroidManifest.xml
...
E: receiver (line=88)
  A: android:name(0x01010003)="com.android.insecurebankv2.MyBroadCastReceiver"␣
↪(Raw: "com.android.insecurebankv2.MyBroadCastReceiver")
  A: android:exported(0x01010010)=(type 0x12)0xffffffff
  E: intent-filter (line=91)
    E: action (line=92)
      A: android:name(0x01010003)="theBroadcast" (Raw: "theBroadcast")
E: receiver (line=119)
  A: android:name(0x01010003)="com.google.android.gms.wallet.
↪EnableWalletOptimizationReceiver" (Raw: "com.google.android.gms.wallet.
↪EnableWalletOptimizationReceiver")
  A: android:exported(0x01010010)=(type 0x12)0x0
  E: intent-filter (line=122)
    E: action (line=123)
      A: android:name(0x01010003)="com.google.android.gms.wallet.ENABLE_WALLET_
↪OPTIMIZATION" (Raw: "com.google.android.gms.wallet.ENABLE_WALLET_OPTIMIZATION")
...
```

You can identify an exported broadcast receiver using one of the following properties:

- It has an intent-filter sub declaration.

- It has the attribute android:exported set to 0xffffffff.

You can also use jadx to identify exported broadcast receivers in the file AndroidManifest.xml using the criteria described above:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.
↪android.insecurebankv2">
...
  <!-- This broadcast receiver is exported via the attribute "exported" as well as␣
↪the "intent-filter" declaration -->
  <receiver android:name="com.android.insecurebankv2.MyBroadCastReceiver"␣
↪android:exported="true">
    <intent-filter>
      <action android:name="theBroadcast"/>
    </intent-filter>
  </receiver>
  <!-- This broadcast receiver is NOT exported because the attribute "exported" is␣
↪explicitly set to false -->
  <receiver android:name="com.google.android.gms.wallet.
↪EnableWalletOptimizationReceiver" android:exported="false">
    <intent-filter>
      <action android:name="com.google.android.gms.wallet.ENABLE_WALLET_
↪OPTIMIZATION"/>
    </intent-filter>
  </receiver>
...
</manifest>
```

The above example from the vulnerable banking application InsecureBankv2 shows that only the broadcast receiver named com.android.insecurebankv2.MyBroadCastReceiver is exported.

Now that you know that there is an exported broadcast receiver, you can dive deeper and reverse engineer the app using jadx. This will allow you to analyze the source code searching for potential vulnerabilities that you could later try to exploit. The source code of the exported broadcast receiver is the following:

```
package com.android.insecurebankv2;
...
public class MyBroadCastReceiver extends BroadcastReceiver {
    public static final String MYPREFS = "mySharedPreferences";
    String usernameBase64ByteString;

    public void onReceive(Context context, Intent intent) {
        String phn = intent.getStringExtra("phonenumber");
        String newpass = intent.getStringExtra("newpass");
        if (phn != null) {
            try {
                SharedPreferences settings = context.getSharedPreferences(
↪"mySharedPreferences", 1);
                this.usernameBase64ByteString = new String(Base64.decode(settings.
↪getString("EncryptedUsername", (String) null), 0), "UTF-8");
                String decryptedPassword = new CryptoClass().
↪aesDeccryptedString(settings.getString("superSecurePassword", (String) null));
                String textPhoneno = phn.toString();
                String textMessage = "Updated Password from: " + decryptedPassword␣
↪+ " to: " + newpass;
                SmsManager smsManager = SmsManager.getDefault();
                System.out.println("For the changepassword - phonenumber: " +␣
↪textPhoneno + " password is: " + textMessage);
                smsManager.sendTextMessage(textPhoneno, (String) null, textMessage,
↪ (PendingIntent) null, (PendingIntent) null);
            } catch (Exception e) {
                e.printStackTrace();
```

(continues on next page)

(continued from previous page)

```
            }
        } else {
            System.out.println("Phone number is null");
        }
    }
}
```

As you can see in the source code, this broadcast receiver expects two parameters named phonenumber and newpass. With this information you can now try to exploit this broadcast receiver by sending events to it using custom values:

```
# Send an event with the following properties:
# Action is set to "theBroadcast"
# Parameter "phonenumber" is set to the string "07123456789"
# Parameter "newpass" is set to the string "12345"
$ adb shell am broadcast -a theBroadcast --es phonenumber "07123456789" --es␣
↪newpass "12345"
Broadcasting: Intent { act=theBroadcast flg=0x400000 (has extras) }
Broadcast completed: result=0
```

This generates the following SMS:

```
Updated Password from: SecretPassword@ to: 12345
```

Sniffing Intents If an Android application broadcasts intents without setting a required permission or specifying the destination package, the intents can be monitored by any application that runs on the device.

To register a broadcast receiver to sniff intents, use the Drozer module app.broadcast.sniff and specify the action to monitor with the – action parameter:

```
dz> run app.broadcast.sniff  --action theBroadcast
[*] Broadcast receiver registered to sniff matching intents
[*] Output is updated once a second. Press Control+C to exit.

Action: theBroadcast
Raw: Intent { act=theBroadcast flg=0x10 (has extras) }
Extra: phonenumber=07123456789 (java.lang.String)
Extra: newpass=12345 (java.lang.String)`
```

You can also use the following command to sniff the intents. However, the content of the extras passed will not be displayed:

```
$ adb shell dumpsys activity broadcasts | grep "theBroadcast"
BroadcastRecord{fc2f46f u0 theBroadcast} to user 0
Intent { act=theBroadcast flg=0x400010 (has extras) }
BroadcastRecord{7d4f24d u0 theBroadcast} to user 0
Intent { act=theBroadcast flg=0x400010 (has extras) }
45: act=theBroadcast flg=0x400010 (has extras)
46: act=theBroadcast flg=0x400010 (has extras)
121: act=theBroadcast flg=0x400010 (has extras)
144: act=theBroadcast flg=0x400010 (has extras)
```

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Sniffing Intents

6.4.4 Rulebook

1. *Use IPC mechanism for security considerations (Required)*

6.4.4.1 Use IPC mechanism for security considerations (Required)

Security considerations and the use of IPC mechanisms prevent the leakage or disclosure of sensitive data from apps.

There are two general methods of disclosing components to other apps Do not publish components that do not need to be published to other (external) apps.

- Set android:exported=" true" in the component definition in AndroidManifest.xml. This exposes the component to other apps.
- Add the definition of <intent-filter> to the component definition in AndroidManifest.xml. If the API level is less than 31, android:exported is automatically set to "true" by defining it. If API level 31 or higher, android:exported must be explicitly set when defining <intent-filter>. If not, the installation will fail.

The following is an example of defining <intent-filter> in AndroidManifest.xml and setting android:exported=" true" .

```
<activity android:name="MyActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>

<service android:name="MyService"
         android:exported="true">
    <intent-filter>
        <action android:name="com.example.app.START_BACKGROUND" />
    </intent-filter>
</service>

<receiver android:name="MyReceiver"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.LOCALE_CHANGED"/>
    </intent-filter>
</receiver>
```

The permission tag (android:permission) can also be used to restrict access to other application components. If IPC is intended to be accessed by other applications, a security policy can be applied with the <permission> element and the appropriate android:protectionLevel can be set.

Below is an example of defining custom permissions in AndroidManifest.xml.

```
<permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY"
        android:label="Start Activity in myapp"
        android:description="Allow the app to launch the activity of myapp app,␣
↪any app you grant this permission will be able to launch main activity by myapp␣
↪app."
        android:protectionLevel="normal" />

<activity android:name="TEST_ACTIVITY"
    android:permission="com.example.myapp.permission.START_MAIN_ACTIVITY">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
     </intent-filter>
</activity>
```

The following is an example of defining custom permissions defined above in AndroidManifest.xml for another app.

```
<manifest>
<uses-permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY" />
        <application>
            <activity>
            </activity>
        </application>
</manifest>
```

If the IPC mechanism needs to be disclosed, make sure the source code does not leak sensitive data when the mechanism is used.

The sample code below shows the process of setting a string to an Intent and sending BroadCast. In these cases, the string should not contain sensitive data.

```
Intent().also { intent ->
    intent.setAction("com.example.broadcast.MY_NOTIFICATION")
    intent.putExtra("data", "Notice me senpai!")
    sendBroadcast(intent)
}
```

If the Intent is broadcast and received only within the application, the LocalBroadcastManager can be used to prevent other apps from receiving the broadcast message. This reduces the risk of sensitive information being compromised.

If this is violated, the following may occur.

- It leads to leakage or exposure of confidential data through IPC mechanisms.

6.5 MSTG-PLATFORM-5

JavaScript is disabled in WebViews unless explicitly required.

6.5.1 Using JavaScript in WebView

JavaScript can be injected into web applications via reflected, stored, or DOM-based Cross-Site Scripting (XSS). Mobile apps are executed in a sandboxed environment and don't have this vulnerability when implemented natively. Nevertheless, WebViews may be part of a native app to allow web page viewing. Every app has its own WebView cache, which isn't shared with the native Browser or other apps. On Android, WebViews use the WebKit rendering engine to display web pages, but the pages are stripped down to minimal functions, for example, pages don't have address bars. If the WebView implementation is too lax and allows usage of JavaScript, JavaScript can be used to attack the app and gain access to its data.

Reference

- owasp-mastg Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5) Overview

6.5.1.1 Static Analysis

The source code must be checked for usage and implementations of the WebView class. To create and use a WebView, you must create an instance of the WebView class.

```
WebView webview = new WebView(this);
setContentView(webview);
webview.loadUrl("https://www.owasp.org/");
```

Various settings can be applied to the WebView (activating/deactivating JavaScript is one example). JavaScript is disabled by default for WebViews and must be explicitly enabled. Look for the method setJavaScriptEnabled to check for JavaScript activation.

```
webview.getSettings().setJavaScriptEnabled(true);
```

This allows the WebView to interpret JavaScript. It should be enabled only if necessary to reduce the attack surface to the app. If JavaScript is necessary, you should make sure that

- The communication to the endpoints consistently relies on HTTPS (or other protocols that allow encryption) to protect HTML and JavaScript from tampering during transmission.
- JavaScript and HTML are loaded locally, from within the app data directory or from trusted web servers only.
- The user cannot define which sources to load by means of loading different resources based on a user provided input.

To remove all JavaScript source code and locally stored data, clear the WebView's cache with clearCache when the app closes.

Devices running platforms older than Android 4.4 (API level 19) use a version of WebKit that has several security issues. As a workaround, the app must confirm that WebView objects display only trusted content if the app runs on these devices.

Reference

- owasp-mastg Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5) Static Analysis

Rulebook

- *Restrict use of JavaScript in WebView (Required)*

6.5.1.2 Dynamic Analysis

Dynamic Analysis depends on operating conditions. There are several ways to inject JavaScript into an app's WebView:

- Stored Cross-Site Scripting vulnerabilities in an endpoint; the exploit will be sent to the mobile app's WebView when the user navigates to the vulnerable function.
- Attacker takes a man-in-the-middle (MITM) position and tampers with the response by injecting JavaScript.
- Malware tampering with local files that are loaded by the WebView.

To address these attack vectors, check the following:

- All functions offered by the endpoint should be free of stored XSS.
- Only files that are in the app data directory should be rendered in a WebView (see test case "Testing for Local File Inclusion in WebViews").
- The HTTPS communication must be implemented according to best practices to avoid MITM attacks. This means:
  - all communication is encrypted via TLS (see test case "Testing for Unencrypted Sensitive Data on the Network"),
  - the certificate is checked properly (see test case "Testing Endpoint Identify Verification"), and/or
  - the certificate should be pinned (see "Testing Custom Certificate Stores and Certificate Pinning").

Reference

- owasp-mastg Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5) Dynamic Analysis

Rulebook

- *Address attack vectors when using JavaScript in WebView (Required)*

6.5.2 Rulebook

1. *Restrict use of JavaScript in WebView (Required)*
2. *Address attack vectors when using JavaScript in WebView (Required)*

6.5.2.1 Restrict use of JavaScript in WebView (Required)

Every app has its own WebView cache, which is not shared with native browsers or other apps. On Android, WebView uses the WebKit rendering engine to display web pages, but the page has minimal functionality, including no address bar. If WebView is poorly implemented and JavaScript is allowed, it is possible to use JavaScript to attack the application and access its data.

Therefore, to use WebView more safely, the use of JavaScript must be restricted.

JavaScript is disabled by default in WebView and must be explicitly enabled in order to use it. To enable it, use the setJavaScriptEnabled method and set the argument to true.

```
WebView webview = new WebView(this);
webview.getSettings().setJavaScriptEnabled(true);
setContentView(webview);
webview.loadUrl("https://www.owasp.org/");
```

Therefore, if you do not need to enable JavaScript, do not set true in the setJavaScriptEnabled method.

If JavaScript is required, the following must be verified.

- Communication to the endpoint consistently relies on HTTPS (or other protocol capable of encryption) to protect HTML and JavaScript from tampering in transit.
- JavaScript and HTML are loaded locally only from within the app' s data directory or from a trusted web server.
- A means of loading different resources based on user-provided input, but not allowing the user to define which sources to load.

If you want to delete all JavaScript source code and locally stored data, use clearCache to clear WebView' s cache when the app exits.

Below is a sample code for clearing WebView' s cache with clearCache.

```
webView.clearCache(true);
```

Devices running platforms older than Android 4.4 (API level 19) are using a version of WebKit that has several security issues. As a workaround, when running your app on these devices, you need to make sure that your app checks that the WebView object shows only trusted content, you need to make sure that your app does so.

If this is violated, the following may occur.

- An app can be attacked using JavaScript to access data within the app.

6.5.2.2 Address attack vectors when using JavaScript in WebView (Required)

To deal with attack vectors that inject JavaScript into an app' s WebView, take the following actions.

- All functions offered by the endpoint should be free of stored XSS.
- Only files that are in the app data directory should be rendered in a WebView.The sample code below shows how to load a file in the application' s data directory in WebView.

  ```
  WebView = new WebView(this);
  webView.loadUrl("file:///android_asset/filename.html");
  ```

- The HTTPS communication must be implemented according to best practices to avoid MITM attacks. This means that.

- all communication is encrypted via TLS. The sample code below is a TLS communication process.

```
    // Load CAs from an InputStream
    // (could be from a resource or ByteArrayInputStream or ...)
    CertificateFactory cf = CertificateFactory.getInstance("X.509");
    // From https://www.washington.edu/itconnect/security/ca/load-der.crt
    InputStream caInput = new BufferedInputStream(new FileInputStream(
↪"load-der.crt"));
    Certificate ca;
    try {
        ca = cf.generateCertificate(caInput);
        System.out.println("ca=" + ((X509Certificate) ca).getSubjectDN());
    } finally {
        caInput.close();
    }

    // Create a KeyStore containing our trusted CAs
    String keyStoreType = KeyStore.getDefaultType();
    KeyStore keyStore = KeyStore.getInstance(keyStoreType);
    keyStore.load(null, null);
    keyStore.setCertificateEntry("ca", ca);

    // Create a TrustManager that trusts the CAs in our KeyStore
    String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
    TrustManagerFactory tmf = TrustManagerFactory.
↪getInstance(tmfAlgorithm);
    tmf.init(keyStore);

    // Create an SSLContext that uses our TrustManager
    SSLContext context = SSLContext.getInstance("TLS");
    context.init(null, tmf.getTrustManagers(), null);

    // Tell the URLConnection to use a SocketFactory from our SSLContext
    URL url = new URL("https://certs.cac.washington.edu/CAtest/");
    HttpsURLConnection urlConnection =
        (HttpsURLConnection)url.openConnection();
    urlConnection.setSSLSocketFactory(context.getSocketFactory());
    InputStream in = urlConnection.getInputStream();
    copyInputStreamToOutputStream(in, System.out);
```

- the certificate is checked properly. See "*Verification by TrustManager (Required)*" for sample code for proper checking of certificates.
- the certificate should be pinned.

If this is violated, the following may occur.

- Stored Cross-Site Scripting vulnerabilities in an endpoint; the exploit will be sent to the mobile app' s WebView when the user navigates to the vulnerable function.
- Attacker takes a man-in-the-middle (MITM) position and tampers with the response by injecting JavaScript.
- Malware tampering with local files that are loaded by the WebView.

6.6 MSTG-PLATFORM-6

WebViews are configured to allow only the minimum set of protocol handlers required (ideally, only https is supported). Potentially dangerous handlers, such as file, tel and app-id, are disabled.

6.6.1 Protocol handlers allowed in WebView

Several default schemas are available for Android URLs. They can be triggered within a WebView with the following:

- http(s)://
- file://
- tel://

WebViews can load remote content from an endpoint, but they can also load local content from the app data directory or external storage. If the local content is loaded, the user shouldn't be able to influence the filename or the path used to load the file, and users shouldn't be able to edit the loaded file.

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Overview

Rulebook

- *WebView does not use potentially dangerous protocol handlers (Recommended)*

6.6.2 Static Analysis

Check the source code for WebView usage. The following WebView settings control resource access:

- setAllowContentAccess: Content URL access allows WebViews to load content from a content provider installed on the system, which is enabled by default .
- setAllowFileAccess: Enables and disables file access within a WebView. The default value is true when targeting Android 10 (API level 29) and below and false for Android 11 (API level 30) and above. Note that this enables and disables file system access only. Asset and resource access is unaffected and accessible via file:///android_asset and file:///android_res.
- setAllowFileAccessFromFileURLs: Does or does not allow JavaScript running in the context of a file scheme URL to access content from other file scheme URLs. The default value is true for Android 4.0.3 - 4.0.4 (API level 15) and below and false for Android 4.1 (API level 16) and above.
- setAllowUniversalAccessFromFileURLs: Does or does not allow JavaScript running in the context of a file scheme URL to access content from any origin. The default value is true for Android 4.0.3 - 4.0.4 (API level 15) and below and false for Android 4.1 (API level 16) and above.

If one or more of the above methods is/are activated, you should determine whether the method(s) is/are really necessary for the app to work properly.

If a WebView instance can be identified, find out whether local files are loaded with the loadURL method.

```
WebView = new WebView(this);
webView.loadUrl("file:///android_asset/filename.html");
```

The location from which the HTML file is loaded must be verified. If the file is loaded from external storage, for example, the file is readable and writable by everyone. This is considered a bad practice. Instead, the file should be placed in the app's assets directory.

```
webview.loadUrl("file:///" +
Environment.getExternalStorageDirectory().getPath() +
"filename.html");
```

The URL specified in loadURL should be checked for dynamic parameters that can be manipulated; their manipulation may lead to local file inclusion.

Use the following code snippet and best practices to deactivate protocol handlers, if applicable:

```
//If attackers can inject script into a WebView, they could access local resources.
↪ This can be prevented by disabling local file system access, which is enabled␣
↪by default. You can use the Android WebSettings class to disable local file␣
↪system access via the public method `setAllowFileAccess`.
webView.getSettings().setAllowFileAccess(false);

webView.getSettings().setAllowFileAccessFromFileURLs(false);

webView.getSettings().setAllowUniversalAccessFromFileURLs(false);

webView.getSettings().setAllowContentAccess(false);
```

- Create a list that defines local and remote web pages and protocols that are allowed to be loaded.
- Create checksums of the local HTML/JavaScript files and check them while the app is starting up. Minify JavaScript files to make them harder to read.

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Static Analysis

Rulebook

- *WebView does not use potentially dangerous protocol handlers (Recommended)*

6.6.3 Dynamic Analysis

To identify the usage of protocol handlers, look for ways to trigger phone calls and ways to access files from the file system while you're using the app.

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Dynamic Analysis

6.6.4 Rulebook

1. *WebView does not use potentially dangerous protocol handlers (Recommended)*

6.6.4.1 WebView does not use potentially dangerous protocol handlers (Recommended)

WebView can load local content from the app's data directory or external storage. Therefore, if an attacker is able to insert scripts into WebView, he/she could gain access to local resources, resulting in the possibility of loading malicious files in WebView. To address this, restrict the use of protocol handlers in WebView to prevent loading local content.Therefore, if loading of local content is not required, it should be set to disable loading.

The following is an example of a configuration that restricts the use of protocol handlers in WebView and prevents local content from being loaded.

- setAllowContentAccess : Content URL access allows WebView to load content from content providers installed on the system, which is enabled by default.
- setAllowFileAccess : Allow URL access to content. Enables or disables file access within WebView. The default value is true if targeting Android 10 (API level 29) or lower, and false if targeting Android 11 (API level 30) or higher. Note that this only enables or disables file system access. Asset and resource access is unaffected and can be accessed via file:///android_asset and file:///android_res.

- setAllowFileAccessFromFileURLs : Specifies whether JavaScript running in the context of a file scheme URL is allowed to access content from other file scheme URLs. The default value is true for Android 4.0.3 - 4.0.4 (API level 15) and below, and false for Android 4.1 (API level 16) and above.
- setAllowUniversalAccessFromFileURLs. Specifies whether JavaScript running in the context of a file scheme URL is allowed to access content from any origin. The default value is true for Android 4.0.3 - 4.0.4 (API level 15) and below, and false for Android 4.1 (API level 16) and above.

The sample code below is an example of processing a configuration that prevents local content from loading in WebView.

```
//If attackers can inject script into a WebView, they could access local resources.
↪ This can be prevented by disabling local file system access, which is enabled␣
↪by default. You can use the Android WebSettings class to disable local file␣
↪system access via the public method `setAllowFileAccess`.
webView.getSettings().setAllowFileAccess(false);

webView.getSettings().setAllowFileAccessFromFileURLs(false);

webView.getSettings().setAllowUniversalAccessFromFileURLs(false);

webView.getSettings().setAllowContentAccess(false);
```

In addition, the following best practices should be followed.

- Create a list defining local and remote web pages and protocols that are allowed to be loaded.
- Create a checksum of local HTML/JavaScript files and check them during app startup. Minimize JavaScript files to make them less readable.

If loading from local storage is required, the file content to be loaded should be placed in the app' s assets directory. This is because file contents in external storage are readable and writable by anyone.

The sample code below is an example of loading file content from external storage that should be avoided. As such, the URL specified in loadURL should be checked for the presence of dynamic parameters that can be manipulated. If present, inclusion of local files that can be modified by the user may occur.

```
webview.loadUrl("file:///" +
Environment.getExternalStorageDirectory().getPath() +
"filename.html");
```

If this is violated, the following may occur.

- A malicious file can be loaded by a WebView to inject scripts into the WebView.

6.7 MSTG-PLATFORM-7

If native methods of the app are exposed to a WebView, verify that the WebView only renders JavaScript contained within the app package.

6.7.1 Bridge between JavaScript and native in WebView

Android offers a way for JavaScript executed in a WebView to call and use native functions of an Android app (annotated with @JavascriptInterface) by using the addJavascriptInterface method. This is known as a WebView JavaScript bridge or native bridge.

Please note that when you use addJavascriptInterface, you' re explicitly granting access to the registered JavaScript Interface object to all pages loaded within that WebView. This implies that, if the user navigates outside your app or domain, all other external pages will also have access to those JavaScript Interface objects which might present a potential security risk if any sensitive data is being exposed though those interfaces.

Warning: Take extreme care with apps targeting Android versions below Android 4.2 (API level 17) as they are vulnerable to a flaw in the implementation of addJavascriptInterface: an attack that is abusing reflection, which leads to remote code execution when malicious JavaScript is injected into a WebView. This was due to all Java Object methods being accessible by default (instead of only those annotated).

Reference

- owasp-mastg Determining Whether Java Objects Are Exposed Through WebViews (MSTG-PLATFORM-7) Overview

Rulebook

- *Be aware of the use of WebView JavaScript Bridge (Required)*

6.7.1.1 Static Analysis

You need to determine whether the method addJavascriptInterface is used, how it is used, and whether an attacker can inject malicious JavaScript.

The following example shows how addJavascriptInterface is used to bridge a Java Object and JavaScript in a WebView:

```
WebView webview = new WebView(this);
WebSettings webSettings = webview.getSettings();
webSettings.setJavaScriptEnabled(true);

MSTG_ENV_008_JS_Interface jsInterface = new MSTG_ENV_008_JS_Interface(this);

myWebView.addJavascriptInterface(jsInterface, "Android");
myWebView.loadURL("http://example.com/file.html");
setContentView(myWebView);
```

In Android 4.2 (API level 17) and above, an annotation @JavascriptInterface explicitly allows JavaScript to access a Java method.

```
public class MSTG_ENV_008_JS_Interface {

        Context mContext;

        /** Instantiate the interface and set the context */
        MSTG_ENV_005_JS_Interface(Context c) {
            mContext = c;
        }

        @JavascriptInterface
        public String returnString () {
            return "Secret String";
        }

        /** Show a toast from the web page */
        @JavascriptInterface
        public void showToast(String toast) {
            Toast.makeText(mContext, toast, Toast.LENGTH_SHORT).show();
        }
}
```

This is how you can call the method returnString from JavaScript, the string "Secret String" will be stored in the variable result:

```
var result = window.Android.returnString();
```

With access to the JavaScript code, via, for example, stored XSS or a MITM attack, an attacker can directly call the exposed Java methods.

If addJavascriptInterface is necessary, take the following considerations:

- Only JavaScript provided with the APK should be allowed to use the bridges, e.g. by verifying the URL on each bridged Java method (via WebView.getUrl).
- No JavaScript should be loaded from remote endpoints, e.g. by keeping page navigation within the app's domains and opening all other domains on the default browser (e.g. Chrome, Firefox).
- If necessary for legacy reasons (e.g. having to support older devices), at least set the minimal API level to 17 in the manifest file of the app (<uses-sdk android:minSdkVersion="17" />).

Reference

- owasp-mastg Determining Whether Java Objects Are Exposed Through WebViews (MSTG-PLATFORM-7) Static Analysis

Rulebook

- *Be aware of the use of WebView JavaScript Bridge (Required)*

6.7.1.2 Dynamic Analysis

Dynamic analysis of the app can show you which HTML or JavaScript files are loaded and which vulnerabilities are present. The procedure for exploiting the vulnerability starts with producing a JavaScript payload and injecting it into the file that the app is requesting. The injection can be accomplished via a MITM attack or direct modification of the file if it is stored in external storage. The whole process can be accomplished via Drozer and weasel (MWR's advanced exploitation payload), which can install a full agent, injecting a limited agent into a running process or connecting a reverse shell as a Remote Access Tool (RAT).

A full description of the attack is included in the blog article by MWR.

Reference

- owasp-mastg Determining Whether Java Objects Are Exposed Through WebViews (MSTG-PLATFORM-7) Dynamic Analysis

6.7.2 Rulebook

1. *Be aware of the use of WebView JavaScript Bridge (Required)*

6.7.2.1 Be aware of the use of WebView JavaScript Bridge (Required)

A method is provided for JavaScript executed in a WebView to call and use native Android app functions (annotated with @JavascriptInterface) using the addJavascriptInterface method.

Note that using @addJavascriptInterface will explicitly grant access to the registered JavaScript Interface object to all pages loaded within that WebView.

Apps targeting Android versions below Android 4.2 (API level 17) have a flaw in the implementation of addJavascript-Interface that could lead to remote code execution if malicious JavaScript is injected into the WebView through a reflection exploit. This can lead to remote code execution if malicious JavaScript is injected into WebView through a reflection attack.

The sample code below is an example of WebView JavaScript Bridge using the addJavascriptInterface method under Android 4.2 (API level 17).

```
WebView webview = new WebView(this);
WebSettings webSettings = webview.getSettings();
webSettings.setJavaScriptEnabled(true);

MSTG_ENV_008_JS_Interface jsInterface = new MSTG_ENV_008_JS_Interface(this);

myWebView.addJavascriptInterface(jsInterface, "Android");
myWebView.loadURL("http://example.com/file.html");
setContentView(myWebView);
```

In Android 4.2 (API level 17) and later, the @JavascriptInterface annotation can be used to explicitly allow JavaScript to access Java methods. Sample code is shown below.

```
public class MSTG_ENV_008_JS_Interface {

        Context mContext;

        /** Instantiate the interface and set the context */
        MSTG_ENV_005_JS_Interface(Context c) {
            mContext = c;
        }

        @JavascriptInterface
        public String returnString () {
            return "Secret String";
        }

        /** Show a toast from the web page */
        @JavascriptInterface
        public void showToast(String toast) {
            Toast.makeText(mContext, toast, Toast.LENGTH_SHORT).show();
        }
}
```

As shown above, when the returnString method is called from JavaScript, the string "Secret String" is stored in the variable "result.

```
var result = window.Android.returnString();
```

If addJavascriptInterface is required, it should be used with the following points in mind

- Only JavaScript provided with the APK should be allowed to use the bridge. For example, validate the URL of each bridged Java method (using WebView.getUrl).
- Do not load JavaScript from remote endpoints. For example, keep page navigation within the app' s domain and open other domains in the default browser (Chrome, Firefox, etc.).
- If required for legacy reasons (e.g. need to support older devices), at least set the minimum API level to 17 in the app' s manifest file (<uses-sdk android:minSdkVersion=" 17" />).

If this is violated, the following may occur.

- May be accessed by JavaScript that does not expect a registered JavaScript Interface object.

6.8 MSTG-PLATFORM-8

Object deserialization, if any, is implemented using safe serialization APIs.

6.8.1 Object Serialization in Android

There are several ways to persist an object on Android:

Reference

- Testing Object Persistence (MSTG-PLATFORM-8) Overview

6.8.1.1 Object Serialization

An object and its data can be represented as a sequence of bytes. This is done in Java via object serialization. Serialization is not inherently secure. It is just a binary format (or representation) for locally storing data in a .ser file. Encrypting and signing HMAC-serialized data is possible as long as the keys are stored safely. Deserializing an object requires a class of the same version as the class used to serialize the object. After classes have been changed, the ObjectInputStream can't create objects from older .ser files. The example below shows how to create a Serializable class by implementing the Serializable interface.

```
import java.io.Serializable;

public class Person implements Serializable {
  private String firstName;
  private String lastName;

  public Person(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    }
  //..
  //getters, setters, etc
  //..

}
```

Now you can read/write the object with ObjectInputStream/ObjectOutputStream in another class.

Reference

- Testing Object Persistence (MSTG-PLATFORM-8) Object Serialization

6.8.1.2 JSON

There are several ways to serialize the contents of an object to JSON. Android comes with the JSONObject and JSONArray classes. A wide variety of libraries, including GSON, Jackson, Moshi, can also be used. The main differences between the libraries are whether they use reflection to compose the object, whether they support annotations, whether the create immutable objects, and the amount of memory they use. Note that almost all the JSON representations are String-based and therefore immutable. This means that any secret stored in JSON will be harder to remove from memory. JSON itself can be stored anywhere, e.g., a (NoSQL) database or a file. You just need to make sure that any JSON that contains secrets has been appropriately protected (e.g., encrypted/HMACed). See the chapter "Data Storage on Android" for more details. A simple example (from the GSON User Guide) of writing and reading JSON with GSON follows. In this example, the contents of an instance of the BagOfPrimitives is serialized into JSON:

```
class BagOfPrimitives {
  private int value1 = 1;
  private String value2 = "abc";
  private transient int value3 = 3;
  BagOfPrimitives() {
    // no-args constructor
  }
}

// Serialization
BagOfPrimitives obj = new BagOfPrimitives();
Gson gson = new Gson();
String json = gson.toJson(obj);

// ==> json is {"value1":1,"value2":"abc"}
```

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) JSON

Rulebook

- *When serializing in JSON/XML format, implement and store additional protections for information that should not be confidential or altered (Required)*

6.8.1.3 XML

There are several ways to serialize the contents of an object to XML and back. Android comes with the XmlPullParser interface which allows for easily maintainable XML parsing. There are two implementations within Android: KXmlParser and ExpatPullParser. The Android Developer Guide provides a great write-up on how to use them. Next, there are various alternatives, such as a SAX parser that comes with the Java runtime. For more information, see a blogpost from ibm.com. Similarly to JSON, XML has the issue of working mostly String based, which means that String-type secrets will be harder to remove from memory. XML data can be stored anywhere (database, files), but do need additional protection in case of secrets or information that should not be changed. See the chapter "Data Storage on Android" for more details. As stated earlier: the true danger in XML lies in the XML eXternal Entity (XXE) attack as it might allow for reading external data sources that are still accessible within the application.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) XML

Rulebook

- *When serializing in JSON/XML format, implement and store additional protections for information that should not be confidential or altered (Required)*
- *Parser application is configured to refuse to resolve external entities (Required)*

6.8.1.4 ORM

There are libraries that provide functionality for directly storing the contents of an object in a database and then instantiating the object with the database contents. This is called Object-Relational Mapping (ORM). Libraries that use the SQLite database include

- OrmLite,
- SugarORM,
- GreenDAO and
- ActiveAndroid.

Realm, on the other hand, uses its own database to store the contents of a class. The amount of protection that ORM can provide depends primarily on whether the database is encrypted. See the chapter "Data Storage on Android" for more details. The Realm website includes a nice example of ORM Lite.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) ORM

6.8.1.5 Parcelable

Parcelable is an interface for classes whose instances can be written to and restored from a Parcel. Parcels are often used to pack a class as part of a Bundle for an Intent. Here's an Android developer documentation example that implements Parcelable:

```
public class MyParcelable implements Parcelable {
    private int mData;

    public int describeContents() {
        return 0;
    }

    public void writeToParcel(Parcel out, int flags) {
```

(continues on next page)

(continued from previous page)

```
        out.writeInt(mData);
    }

    public static final Parcelable.Creator<MyParcelable> CREATOR
            = new Parcelable.Creator<MyParcelable>() {
        public MyParcelable createFromParcel(Parcel in) {
            return new MyParcelable(in);
        }

        public MyParcelable[] newArray(int size) {
            return new MyParcelable[size];
        }
    };

    private MyParcelable(Parcel in) {
        mData = in.readInt();
    }
}
```

Because this mechanism that involves Parcels and Intents may change over time, and the Parcelable may contain IBinder pointers, storing data to disk via Parcelable is not recommended.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Parcelable

Rulebook

- *Do not store data on disk via Parcelable (Recommended)*

6.8.1.6 Protocol Buffers

Protocol Buffers by Google, are a platform- and language neutral mechanism for serializing structured data by means of the Binary Data Format. There have been a few vulnerabilities with Protocol Buffers, such as CVE-2015-5237. Note that Protocol Buffers do not provide any protection for confidentiality: there is no built in encryption.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Protocol Buffers

Rulebook

- *Avoid serialization in Protocol Buffers (Recommended)*

6.8.2 Static Analysis

If object persistence is used for storing sensitive information on the device, first make sure that the information is encrypted and signed/HMACed. See the chapters "Data Storage on Android" and "Android Cryptographic APIs" for more details. Next, make sure that the decryption and verification keys are obtainable only after the user has been authenticated. Security checks should be carried out at the correct positions, as defined in best practices.

There are a few generic remediation steps that you can always take:

1. Make sure that sensitive data has been encrypted and HMACed/signed after serialization/persistence. Evaluate the signature or HMAC before you use the data. See the chapter "Android Cryptographic APIs" for more details.
2. Make sure that the keys used in step 1 can't be extracted easily. The user and/or application instance should be properly authenticated/authorized to obtain the keys. See the chapter "Data Storage on Android" for more details.
3. Make sure that the data within the de-serialized object is carefully validated before it is actively used (e.g., no exploit of business/application logic).

For high-risk applications that focus on availability, we recommend that you use Serializable only when the serialized classes are stable. Second, we recommend not using reflection-based persistence because

- the attacker could find the method' s signature via the String-based argument
- the attacker might be able to manipulate the reflection-based steps to execute business logic.

See the chapter "Android Anti-Reversing Defenses" for more details.

Reference

- owasp- mastg Testing Object Persistence (MSTG-PLATFORM-8) Static Analysis

Rulebook

- *When serializing in JSON/XML format, implement and store additional protections for information that should not be confidential or altered (Required)*
- *If object persistence is used to store sensitive information on the device, the information is encrypted and signed/HMAC (Required)*
- *For high-risk applications that care about availability, use Serializable only if the serialized classes are stable (Recommended)*

6.8.2.1 Object Serialization

Search the source code for the following keywords:

- import java.io.Serializable
- implements Serializable

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Object Serialization

6.8.2.2 JSON

If you need to counter memory-dumping, make sure that very sensitive information is not stored in the JSON format because you can' t guarantee prevention of anti-memory dumping techniques with the standard libraries. You can check for the following keywords in the corresponding libraries:

JSONObject Search the source code for the following keywords:

- import org.json.JSONObject;
- import org.json.JSONArray;

GSON Search the source code for the following keywords:

- import com.google.gson
- import com.google.gson.annotations
- import com.google.gson.reflect
- import com.google.gson.stream
- new Gson();
- Annotations such as @Expose, @JsonAdapter, @SerializedName,@Since, and @Until

Jackson Search the source code for the following keywords:

- import com.fasterxml.jackson.core
- import org.codehaus.jackson for the older version.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) JSON

Rulebook

- *Do not store very important information in JSON format (Required)*

6.8.2.3 ORM

When you use an ORM library, make sure that the data is stored in an encrypted database and the class representations are individually encrypted before storing it. See the chapters "Data Storage on Android" and "Android Cryptographic APIs" for more details. You can check for the following keywords in the corresponding libraries:

OrmLite Search the source code for the following keywords:

- import com.j256.*
- import com.j256.dao
- import com.j256.db
- import com.j256.stmt
- import com.j256.table\

Please make sure that logging is disabled.

SugarORM Search the source code for the following keywords:

- import com.github.satyan
- extends SugarRecord<Type>
- In the AndroidManifest, there will be meta-data entries with values such as DATABASE, VERSION, QUERY_LOG and DOMAIN_PACKAGE_NAME.

Make sure that QUERY_LOG is set to false.

GreenDAO Search the source code for the following keywords:

- import org.greenrobot.greendao.annotation.Convert
- import org.greenrobot.greendao.annotation.Entity
- import org.greenrobot.greendao.annotation.Generated
- import org.greenrobot.greendao.annotation.Id
- import org.greenrobot.greendao.annotation.Index
- import org.greenrobot.greendao.annotation.NotNull
- import org.greenrobot.greendao.annotation.*
- import org.greenrobot.greendao.database.Database
- import org.greenrobot.greendao.query.Query

ActiveAndroid Search the source code for the following keywords:

- ActiveAndroid.initialize (<contextReference>);
- import com.activeandroid.Configuration
- import com.activeandroid.query.*

Realm Search the source code for the following keywords:

- import io.realm.RealmObject;
- import io.realm.annotations.PrimaryKey;

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) ORM

Rulebook

- *Notes on serialization in ORM (Required)*

6.8.2.4 Parcelable

Make sure that appropriate security measures are taken when sensitive information is stored in an Intent via a Bundle that contains a Parcelable. Use explicit Intents and verify proper additional security controls when using application-level IPC (e.g., signature verification, intent-permissions, crypto).

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Parcelable

Rulebook

- *Take appropriate security measures when sensitive information is stored in an Intent via a Bundle containing a Parcelable (Required)*

6.8.3 Dynamic Analysis

There are several ways to perform dynamic analysis:

1. For the actual persistence: Use the techniques described in the data storage chapter.
2. For reflection-based approaches: Use Xposed to hook into the deserialization methods or add unprocessable information to the serialized objects to see how they are handled (e.g., whether the application crashes or extra information can be extracted by enriching the objects).

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Dynamic Analysis

6.8.4 Rulebook

1. *When serializing in JSON/XML format, implement and store additional protections for information that should not be confidential or altered (Required)*
2. *Do not store data on disk via Parcelable (Recommended)*
3. *Avoid serialization in Protocol Buffers (Recommended)*
4. *If object persistence is used to store sensitive information on the device, the information is encrypted and signed/HMAC (Required)*
5. *For high-risk applications that care about availability, use Serializable only if the serialized classes are stable (Recommended)*
6. *Do not store very important information in JSON format (Required)*
7. *Notes on serialization in ORM (Required)*
8. *Take appropriate security measures when sensitive information is stored in an Intent via a Bundle containing a Parcelable (Required)*

6.8.4.1 When serializing in JSON/XML format, implement and store additional protections for information that should not be confidential or altered (Required)

JSON/XML data can be stored anywhere. Therefore, due to the risk of access to files stored in external storage, it is necessary to consider the storage location and encryption when saving information that should not be secret or changed.

See the rule below for how to save data to internal storage and a sample code to save data with encryption.

Rulebook

- *Store sensitive data in the app container or in the system's credentials storage function (Required)*
- *Key material must be erased from memory as soon as it is no longer needed (Required)*

If this is violated, the following may occur.

- Files stored in external storage may be accessed and leaked.

6.8.4.2 Do not store data on disk via Parcelable (Recommended)

Because the mechanism involving Parcel and Intent may change over time, and because Parcelable may contain an IBinder pointer, storing data to disk via Parcelable is not recommended.

The sample code below is an example implementation of the Parcelable interface.

```
public class MyParcelable implements Parcelable {
     private int mData;

     public int describeContents() {
         return 0;
     }

     public void writeToParcel(Parcel out, int flags) {
         out.writeInt(mData);
     }

     public static final Parcelable.Creator<MyParcelable> CREATOR
             = new Parcelable.Creator<MyParcelable>() {
         public MyParcelable createFromParcel(Parcel in) {
             return new MyParcelable(in);
         }

         public MyParcelable[] newArray(int size) {
             return new MyParcelable[size];
         }
     };

     private MyParcelable(Parcel in) {
         mData = in.readInt();
     }
 }
```

If this is not noted, the following may occur.

- May contain IBinder pointers.

6.8.4.3 Avoid serialization in Protocol Buffers (Recommended)

Protocol Buffers by Google, are a platform- and language neutral mechanism for serializing structured data by means of the Binary Data Format. There have been a few vulnerabilities with Protocol Buffers, such as CVE-2015-5237. Note that Protocol Buffers do not provide any protection for confidentiality: there is no built in encryption.Therefore, it should not be used as a secure implementation.

* There is no sample code because it is a deprecated rule.

If this is not noted, the following may occur.

- An attacker could cause a heap-based buffer overflow.
- Confidential data can be read by third parties.

6.8.4.4 If object persistence is used to store sensitive information on the device, the information is encrypted and signed/HMAC (Required)

If object persistence is used for storing sensitive information on the device, first make sure that the information is encrypted and signed/HMACed. If not, perform the following general remediation steps.

1. Make sure that sensitive data has been encrypted and HMACed/signed after serialization/persistence. Evaluate the signature or HMAC before you use the data. See the chapter "Android Cryptographic APIs" for more details.
2. Make sure that the keys used in step 1 can' t be extracted easily. The user and/or application instance should be properly authenticated/authorized to obtain the keys. See the chapter "Data Storage on Android" for more details.
3. Make sure that the data within the de-serialized object is carefully validated before it is actively used (e.g., no exploit of business/application logic).

If this is violated, the following may occur.

- Sensitive data stored by object serialization can be read by a third party.

6.8.4.5 For high-risk applications that care about availability, use Serializable only if the serialized classes are stable (Recommended)

For high-risk applications that focus on availability, we recommend that you use Serializable only when the serialized classes are stable. Second, we recommend not using reflection-based persistence because.

* There is no sample code because it is a conceptual rule.

If this is not noted, the following may occur.

- Attackers can discover method signatures through String-based arguments.
- Attackers can manipulate reflection-based steps to execute business logic.

6.8.4.6 Do not store very important information in JSON format (Required)

When memory dump countermeasures are required, very important information is not stored in JSON format because the standard library cannot guarantee the prevention of memory dump countermeasure methods. The following keywords can be checked in the corresponding library.

JSONObject Search the source code for the following keywords:

- import org.json.JSONObject;
- import org.json.JSONArray;

GSON Search the source code for the following keywords:

- import com.google.gson
- import com.google.gson.annotations

- import com.google.gson.reflect
- import com.google.gson.stream
- new Gson();
- Annotations such as @Expose, @JsonAdapter, @SerializedName,@Since, and @Until

Jackson Search the source code for the following keywords:

- import com.fasterxml.jackson.core
- import org.codehaus.jackson for the older version.

* No sample code due to deprecated rules.

If this is violated, the following may occur.

- A memory dump can leak information.

6.8.4.7 Notes on serialization in ORM (Required)

Note the following when serializing in ORM.

- When you use an ORM library, make sure that the data is stored in an encrypted database and the class representations are individually encrypted before storing it. See "*SQLite*" for sample code for encrypted databases.
- Please make sure that logging is disabled.

If this is violated, the following may occur.

- A third party may read the stored confidential data.
- A third party may be able to read confidential data from the contents of the log.

6.8.4.8 Take appropriate security measures when sensitive information is stored in an Intent via a Bundle containing a Parcelable (Required)

Make sure that appropriate security measures are taken when sensitive information is stored in an Intent via a Bundle that contains a Parcelable. Use explicit Intents and verify proper additional security controls when using application-level IPC (e.g., signature verification, intent-permissions, crypto).

For security measures and sample code when using IPC mechanisms and Intents, see "*Use IPC mechanism for security considerations (Required)*" .

If this is violated, the following may occur.

- Sensitive data can be read by other apps through PC mechanisms.

7 Code Quality and Build Setting Requirements

7.1 MSTG-CODE-1

The app is signed and provisioned with a valid certificate, of which the private key is properly protected.

7.1.1 Factors to consider when signing an app

7.1.1.1 Expiration date of the certificate to be used

Android requires all APKs to be digitally signed with a certificate before they are installed or run. The digital signature is used to verify the owner's identity for application updates. This process can prevent an app from being tampered with or modified to include malicious code.

When an APK is signed, a public-key certificate is attached to it. This certificate uniquely associates the APK with the developer and the developer's private key. When an app is being built in debug mode, the Android SDK signs the app with a debug key created specifically for debugging purposes. An app signed with a debug key is not meant to be distributed and won't be accepted in most app stores, including the Google Play Store.

The final release build of an app must be signed with a valid release key. In Android Studio, the app can be signed manually or via creation of a signing configuration that's assigned to the release build type.

Prior Android 9 (API level 28) all app updates on Android need to be signed with the same certificate, so a validity period of 25 years or more is recommended. Apps published on Google Play must be signed with a key that that has a validity period ending after October 22th, 2033.

Reference

- owasp-mastg Making Sure That the App is Properly Signed (MSTG-CODE-1) Overview

Rulebook

- *In the final release build of the app, sign with valid release skis (Required)*
- *Certificate expiration date (Recommended)*

7.1.1.2 Signature Scheme of the Application

Four APK signing schemes are available:

- JAR signing (v1 scheme),
- APK Signature Scheme v2 (v2 scheme),
- APK Signature Scheme v3 (v3 scheme).
- APK Signature Scheme v4 (v4 scheme).

The v2 signature, which is supported by Android 7.0 (API level 24) and above, offers improved security and performance compared to v1 scheme. The V3 signature, which is supported by Android 9 (API level 28) and above, gives apps the ability to change their signing keys as part of an APK update. The V4 signature, which is supported by Android 11 (API level 30) and above. This functionality assures compatibility and apps continuous availability by allowing both the new and the old keys to be used. Note that it is only available via apksigner at the time of writing.

For each signing scheme the release builds should always be signed via all its previous schemes as well.

Reference

- owasp-mastg Making Sure That the App is Properly Signed (MSTG-CODE-1) Overview

Rulebook

- *Improvement of security with the signature scheme of the app (Recommended)*

7.1.2 Static Analysis

Make sure that the release build has been signed via both the v1 and v2 schemes for Android 7.0 (API level 24) and above and via all the three schemes for Android 9 (API level 28) and above, and that the code-signing certificate in the APK belongs to the developer.

APK signatures can be verified with the apksigner tool. It is located at [SDK-Path]/build-tools/[version].

```
$ apksigner verify --verbose Desktop/example.apk
Verifies
Verified using v1 scheme (JAR signing): true
Verified using v2 scheme (APK Signature Scheme v2): true
Verified using v3 scheme (APK Signature Scheme v3): true
Number of signers: 1
```

The contents of the signing certificate can be examined with jarsigner. Note that the Common Name (CN) attribute is set to "Android Debug" in the debug certificate.

The output for an APK signed with a debug certificate is shown below:

```
$ jarsigner -verify -verbose -certs example.apk

sm     11116 Fri Nov 11 12:07:48 ICT 2016 AndroidManifest.xml

      X.509, CN=Android Debug, O=Android, C=US
      [certificate is valid from 3/24/16 9:18 AM to 8/10/43 9:18 AM]
      [CertPath not validated: Path doesn\'t chain with any of the trust anchors]
(...)
```

Ignore the "CertPath not validated" error. This error occurs with Java SDK 7 and above. Instead of jarsigner, you can rely on the apksigner to verify the certificate chain.

The signing configuration can be managed through Android Studio or the signingConfig block in build.gradle. To activate both the v1 and v2 schemes, the following values must be set:

```
v1SigningEnabled true
v2SigningEnabled true
```

Several best practices for configuring the app for release are available in the official Android developer documentation.

Last but not least: make sure that the application is never deployed with your internal testing certificates.

Reference

- owasp-mastg Making Sure That the App is Properly Signed (MSTG-CODE-1) Static Analysis

Rulebook

- *Signs that match the target OS version (Required)*
- *The application will never be deployed with the internal test certificate (Required)*

7.1.3 Dynamic Analysis

Static analysis should be used to verify the APK signature.

Reference

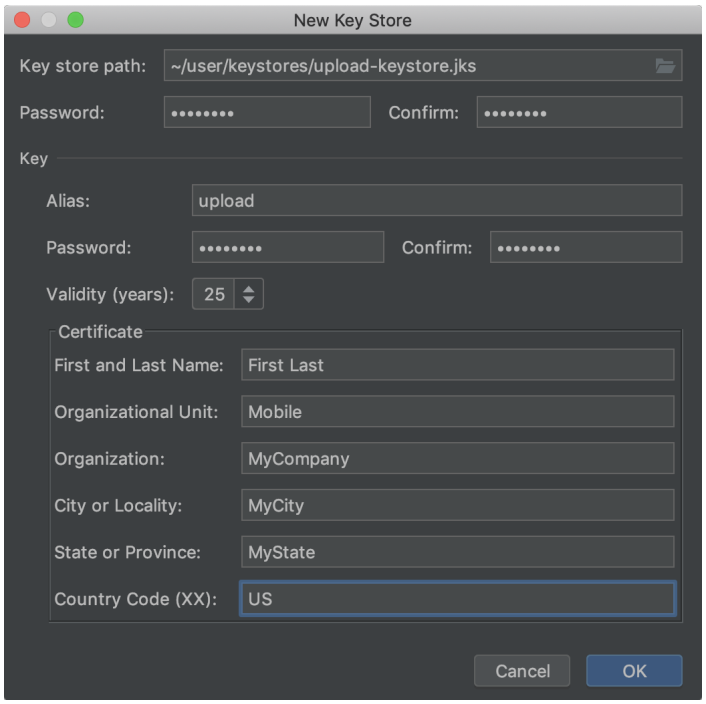
- owasp-mastg Making Sure That the App is Properly Signed (MSTG-CODE-1) Dynamic Analysis

7.1.4 Rulebook

1. *In the final release build of the app, sign with valid release skis (Required)*
2. *Certificate expiration date (Recommended)*
3. *Improvement of security with the signature scheme of the app (Recommended)*
4. *Signs that match the target OS version (Required)*
5. *The application will never be deployed with the internal test certificate (Required)*

7.1.4.1 In the final release build of the app, sign with valid release skis (Required)

The final release build of an app must be signed with a valid release key. In Android Studio, the app can be signed manually or via creation of a signing configuration that's assigned to the release build type.



Reference

- Sign your app for release to Google Play

If this is violated, the following may occur.

- Unable to publish apps on Google Play.

7.1.4.2 Certificate expiration date (Recommended)

Prior Android 9 (API level 28) all app updates on Android need to be signed with the same certificate, so a validity period of 25 years or more is recommended. Apps published on Google Play must be signed with a key that that has a validity period ending after October 22th, 2033.

If this is not noted, the following may occur.

- Once the key expires, users will not be able to seamlessly upgrade to a newer version of the app.

7.1.4.3 Improvement of security with the signature scheme of the app (Recommended)

Four APK signing schemes are available:

- JAR signing (v1 scheme),
- APK Signature Scheme v2 (v2 scheme),
- APK Signature Scheme v3 (v3 scheme).
- APK Signature Scheme v4 (v4 scheme).

The v2 signature, which is supported by Android 7.0 (API level 24) and above, offers improved security and performance compared to v1 scheme. The V3 signature, which is supported by Android 9 (API level 28) and above, gives apps the ability to change their signing keys as part of an APK update. The V4 signature, which is supported by Android 11 (API level 30) and above. This functionality assures compatibility and apps continuous availability by allowing both the new and the old keys to be used. Note that it is only available via apksigner at the time of writing.

For each signing scheme, release builds must always be signed by all previous schemes as well. etc

Here is how to sign with apksigner.

```
apksigner sign --ks keystore.jks |
  --key key.pk8 --cert cert.x509.pem
  [signer_options] app-name.apk


--v1-signing-enabled <true | false>
Specifies whether apksigner should use the traditional JAR-based signature scheme␣
↪when signing the specified APK package. By default, the tool uses the --min-sdk-
↪version and --max-sdk-version values to determine when to apply this signature␣
↪scheme.
--v2-signing-enabled <true | false>
Specifies whether apksigner should use APK Signature Scheme v2 when signing the␣
↪specified APK package. By default, the tool uses the --min-sdk-version and --max-
↪sdk-version values to determine when to apply this signature scheme.
--v3-signing-enabled <true | false>
Specifies whether apksigner should use APK Signature Scheme v3 when signing the␣
↪specified APK package. By default, the tool uses the --min-sdk-version and --max-
↪sdk-version values to determine when to apply this signature scheme.
```

Reference

- JAR signing (v1 scheme)
- APK Signature Scheme v2 (v2 scheme)
- APK Signature Scheme v3 (v3 scheme)
- APK Signature Scheme v4 (v4 scheme)

If this is not noted, the following may occur.

- May degrade security and performance.

7.1.4.4 Signs that match the target OS version (Required)

Make sure that the release build has been signed via both the v1 and v2 schemes for Android 7.0 (API level 24) and above and via all the three schemes (v1, v2, v3) for Android 9 (API level 28) and above, and that the code-signing certificate in the APK belongs to the developer. Android 11 (API level 30) and higher requires a v4 signature and a complementary v2 or v3 signature. To support devices that run older versions of Android, you should continue to sign your APKs using APK Signature Scheme v1, in addition to signing your APK with APK Signature Scheme v2 or higher.

The following is how to sign an APK using the apksigner command.

```
apksigner sign --ks [keystore file] -v --ks-key-alias [key alias] --ks-pass␣
↪pass:[keystore password] [Unsigned APK file]
```

If this is violated, the following may occur.

- May degrade security and performance.

7.1.4.5 The application will never be deployed with the internal test certificate (Required)

Make sure that the application is never deployed with your internal testing certificates.

If this is violated, the following may occur.

- Logging and debugging may be deployed with logging and debugging enabled.
- May not be accepted in the app store.

7.2 MSTG-CODE-2

The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable).

7.2.1 Toggle application debugging enable/disable

The android:debuggable attribute in the Application element that is defined in the Android manifest determines whether the app can be debugged or not.

Reference

- owasp-mastg Testing Whether the App is Debuggable (MSTG-CODE-2) Overview

7.2.2 Static Analysis

Check AndroidManifest.xml to determine whether the android:debuggable attribute has been set and to find the attribute’ s value:

```
    ...
    <application android:allowBackup="true" android:debuggable="true" android:icon=
↪"@drawable/ic_launcher" android:label="@string/app_name" android:theme="@style/
↪AppTheme">
    ...
```

You can use aapt tool from the Android SDK with the following command line to quickly check if the android:debuggable=” true” directive is present:

```
# If the command print 1 then the directive is present
# The regex search for this line: android:debuggable(0x0101000f)=(type␣
↪0x12)0xffffffff
$ aapt d xmltree sieve.apk AndroidManifest.xml | grep -Ec "android:debuggable\
↪(0x[0-9a-f]+\)=\(type\s0x[0-9a-f]+\)0xffffffff"
1
```

For a release build, this attribute should always be set to "false" (the default value).

Reference

- owasp-mastg Testing Whether the App is Debuggable (MSTG-CODE-2) Static Analysis

Rulebook

- *The android:debuggable attribute should be set to false when released (Recommended)*

7.2.3 Dynamic Analysis

adb can be used to determine whether an application is debuggable.

Use the following command:

```
# If the command print a number superior to zero then the application have the␣
↪debug flag
# The regex search for these lines:
# flags=[ DEBUGGABLE HAS_CODE ALLOW_CLEAR_USER_DATA ALLOW_BACKUP ]
# pkgFlags=[ DEBUGGABLE HAS_CODE ALLOW_CLEAR_USER_DATA ALLOW_BACKUP ]
$ adb shell dumpsys package com.mwr.example.sieve | grep -c "DEBUGGABLE"
2
$ adb shell dumpsys package com.nondebuggableapp | grep -c "DEBUGGABLE"
0
```

If an application is debuggable, executing application commands is trivial. In the adb shell, execute run-as by appending the package name and application command to the binary name:

```
$ run-as com.vulnerable.app id
uid=10084(u0_a84) gid=10084(u0_a84) groups=10083(u0_a83),1004(input),1007(log),
↪1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),
↪3003(inet),3006(net_bw_stats) context=u:r:untrusted_app:s0:c512,c768
```

Android Studio can also be used to debug an application and verify debugging activation for an app.

Another method for determining whether an application is debuggable is attaching jdb to the running process. If this is successful, debugging will be activated.

The following procedure can be used to start a debug session with jdb:

1. Using adb and jdwp, identify the PID of the active application that you want to debug:

```
$ adb jdwp
2355
16346  <== last launched, corresponds to our application
```

2. Create a communication channel by using adb between the application process (with the PID) and your host computer by using a specific local port:

```
# adb forward tcp:[LOCAL_PORT] jdwp:[APPLICATION_PID]
$ adb forward tcp:55555 jdwp:16346
```

3. Using jdb, attach the debugger to the local communication channel port and start a debug session:

```
$ jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=55555
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> help
```

A few notes about debugging:

- The tool JADX can be used to identify interesting locations for breakpoint insertion.
- Usage of basic commands for jdb can be found at Tutorialspoint.
- If you get an error telling that "the connection to the debugger has been closed" while jdb is being bound to the local communication channel port, kill all adb sessions and start a single new session.

Reference

- owasp-mastg Testing Whether the App is Debuggable (MSTG-CODE-2) Dynamic Analysis

7.2.4 Rulebook

1. *The android:debuggable attribute should be set to false when released (Recommended)*

7.2.4.1 The android:debuggable attribute should be set to false when released (Recommended)

At the time of release, the android:debuggable attribute is all set to false.

If this is violated, the following may occur.

- It can be misused by malicious users.

7.3 MSTG-CODE-3

Debugging symbols have been removed from native binaries.

7.3.1 Presence or absence of debug symbols

Generally, you should provide compiled code with as little explanation as possible. Some metadata, such as debugging information, line numbers, and descriptive function or method names, make the binary or bytecode easier for the reverse engineer to understand, but these aren't needed in a release build and can therefore be safely omitted without impacting the app's functionality.

To inspect native binaries, use a standard tool like nm or objdump to examine the symbol table. A release build should generally not contain any debugging symbols. If the goal is to obfuscate the library, removing unnecessary dynamic symbols is also recommended.

Reference

- owasp-mastg Testing for Debugging Symbols (MSTG-CODE-3) Overview

Rulebook

- *Not output code information in the release build (Required)*

7.3.2 Static Analysis

Symbols are usually stripped during the build process, so you need the compiled bytecode and libraries to make sure that unnecessary metadata has been discarded.

First, find the nm binary in your Android NDK and export it (or create an alias).

```
export NM = $ANDROID_NDK_DIR/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-
↪x86_64/bin/arm-linux-androideabi-nm
```

To display debug symbols:

```
$NM -a libfoo.so
/tmp/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin/arm-linux-
↪androideabi-nm: libfoo.so: no symbols
```

To display dynamic symbols:

```
$NM -D libfoo.so
```

Alternatively, open the file in your favorite disassembler and check the symbol tables manually.

Dynamic symbols can be stripped via the visibility compiler flag. Adding this flag causes gcc to discard the function names while preserving the names of functions declared as JNIEXPORT.

Make sure that the following has been added to build.gradle:

```
externalNativeBuild {
    cmake {
        cppFlags "-fvisibility=hidden"
    }
}
```

Reference

- owasp-mastg Testing for Debugging Symbols (MSTG-CODE-3) Static Analysis

7.3.3 Dynamic Analysis

Static analysis should be used to verify debugging symbols.

Reference

- owasp-mastg Testing for Debugging Symbols (MSTG-CODE-3) Dynamic Analysis

7.3.4 Rulebook

1. *Not output code information in the release build (Required)*

7.3.4.1 Not output code information in the release build (Required)

Generally, you should provide compiled code with as little explanation as possible. Some metadata, such as debugging information, line numbers, and descriptive function or method names, make the binary or bytecode easier for the reverse engineer to understand, but these aren't needed in a release build and can therefore be safely omitted without impacting the app's functionality.

A release build should generally not contain any debugging symbols. If the goal is to obfuscate the library, removing unnecessary dynamic symbols is also recommended.

Dynamic symbols can be removed by the visibility compiler flag. Adding this flag causes gcc to discard the function name while retaining the name of the function declared as JNIEXPORT.

Make sure the following is added to build.gradle

```
externalNativeBuild {
    cmake {
        cppFlags "-fvisibility=hidden"
    }
}
```

If this is violated, the following may occur.

- Some metadata in the code, such as debugging information, line numbers, descriptive function or method names, etc., may be leaked.

7.4 MSTG-CODE-4

Debugging code and developer assistance code (e.g. test code, backdoors, hidden settings) have been removed. The app does not log verbose errors or debugging messages.

7.4.1 Using StrictMode

StrictMode is a developer tool for detecting violations, e.g. accidental disk or network access on the application' s main thread. It can also be used to check for good coding practices, such as implementing performant code.

Here is an example of StrictMode with policies enabled for disk and network access to the main thread:

```
public void onCreate() {
     if (DEVELOPER_MODE) {
          StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
                  .detectDiskReads()
                  .detectDiskWrites()
                  .detectNetwork()   // or .detectAll() for all detectable problems
                  .penaltyLog()
                  .build());
          StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
                  .detectLeakedSqlLiteObjects()
                  .detectLeakedClosableObjects()
                  .penaltyLog()
                  .penaltyDeath()
                  .build());
     }
     super.onCreate();
 }
```

Inserting the policy in the if statement with the DEVELOPER_MODE condition is recommended. To disable StrictMode, DEVELOPER_MODE must be disabled for the release build.

Reference

- owasp-mastg Testing for Debugging Code and Verbose Error Logging (MSTG-CODE-4) Overview

Rulebook

- *Use StrictMode only when debugging (Recommended)*

7.4.2 Static Analysis

To determine whether StrictMode is enabled, you can look for the StrictMode.setThreadPolicy or StrictMode.setVmPolicy methods. Most likely, they will be in the onCreate method.

The detection methods for the thread policy are

```
detectDiskWrites()
detectDiskReads()
detectNetwork()
```

The penalties for thread policy violation are

```
penaltyLog() // Logs a message to LogCat
penaltyDeath() // Crashes application, runs at the end of all enabled penalties
penaltyDialog() // Shows a dialog
```

Have a look at the best practices for using StrictMode.

Reference

- owasp-mastg Testing for Debugging Code and Verbose Error Logging (MSTG-CODE-4) Static Analysis

7.4.3 Dynamic Analysis

There are several ways of detecting StrictMode; the best choice depends on how the policies' roles are implemented. They include

- Logcat,
- a warning dialog,
- application crash.

Reference

- owasp-mastg Testing for Debugging Code and Verbose Error Logging (MSTG-CODE-4) Dynamic Analysis

7.4.4 Rulebook

1. *Use StrictMode only when debugging (Recommended)*

7.4.4.1 Use StrictMode only when debugging (Recommended)

StrictMode is a developer tool for detecting violations, e.g. accidental disk or network access on the application's main thread. When using StrictMode, thread policy and VM policy must be set. Therefore, it is necessary to take measures such as setting a branch before the policy setting process to prevent the StrictMode setting process incorporated during development from being incorporated into the released version of the application.

The following is a sample code that incorporates StrictMode policy settings for development.

```
public void onCreate() {
     if (DEVELOPER_MODE) {
          StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
                  .detectDiskReads()
                  .detectDiskWrites()
                  .detectNetwork()   // or .detectAll() for all detectable problems
                  .penaltyLog()
                  .build());
          StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
                  .detectLeakedSqlLiteObjects()
```

(continues on next page)

(continued from previous page)

```
                .detectLeakedClosableObjects()
                .penaltyLog()
                .penaltyDeath()
                .build());
    }
    super.onCreate();
}
```

In this case, the following possibilities exist.

- Disk access and other information may be compromised.

7.5 MSTG-CODE-5

All third party components used by the mobile app, such as libraries and frameworks, are identified, and checked for known vulnerabilities.

7.5.1 Checking for Weaknesses in Third Party Libraries

Android apps often make use of third party libraries. These third party libraries accelerate development as the developer has to write less code in order to solve a problem. There are two categories of libraries:

- Libraries that are not (or should not) be packed within the actual production application, such as Mockito used for testing and libraries like JavaAssist used to compile certain other libraries.
- Libraries that are packed within the actual production application, such as Okhttp3.

These libraries can lead to unwanted side-effects:

- A library can contain a vulnerability, which will make the application vulnerable. A good example are the versions of OKHTTP prior to 2.7.5 in which TLS chain pollution was possible to bypass SSL pinning.
- A library can no longer be maintained or hardly be used, which is why no vulnerabilities are reported and/or fixed. This can lead to having bad and/or vulnerable code in your application through the library.
- A library can use a license, such as LGPL2.1, which requires the application author to provide access to the source code for those who use the application and request insight in its sources. In fact the application should then be allowed to be redistributed with modifications to its sourcecode. This can endanger the intellectual property (IP) of the application.

Please note that this issue can hold on multiple levels: When you use webviews with JavaScript running in the webview, the JavaScript libraries can have these issues as well. The same holds for plugins/libraries for Cordova, React-native and Xamarin apps.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Overview

Rulebook

- *Pay attention to the use of third-party libraries (Recommended)*

7.5.2 Static Analysis

7.5.2.1 Vulnerabilities of Third Party Libraries

Detecting vulnerabilities in third party dependencies can be done by means of the OWASP Dependency checker. This is best done by using a gradle plugin, such as dependency-check-gradle. In order to use the plugin, the following steps need to be applied: Install the plugin from the Maven central repository by adding the following script to your build.gradle:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.owasp:dependency-check-gradle:3.2.0'
    }
}

apply plugin: 'org.owasp.dependencycheck'
```

Once gradle has invoked the plugin, you can create a report by running:

```
gradle assemble
gradle dependencyCheckAnalyze --info
```

The report will be in build/reports unless otherwise configured. Use the report in order to analyze the vulnerabilities found. See remediation on what to do given the vulnerabilities found with the libraries.

Please be advised that the plugin requires to download a vulnerability feed. Consult the documentation in case issues arise with the plugin.

Alternatively there are commercial tools which might have a better coverage of the dependencies found for the libraries being used, such as Sonatype Nexus IQ, Sourceclear, Snyk or Blackduck. The actual result of using either the OWASP Dependency Checker or another tool varies on the type of (NDK related or SDK related) libraries.

Lastly, please note that for hybrid applications, one will have to check the JavaScript dependencies with RetireJS. Similarly for Xamarin, one will have to check the C# dependencies.

When a library is found to contain vulnerabilities, then the following reasoning applies:

- Is the library packaged with the application? Then check whether the library has a version in which the vulnerability is patched. If not, check whether the vulnerability actually affects the application. If that is the case or might be the case in the future, then look for an alternative which provides similar functionality, but without the vulnerabilities.
- Is the library not packaged with the application? See if there is a patched version in which the vulnerability is fixed. If this is not the case, check if the implications of the vulnerability for the build-process. Could the vulnerability impede a build or weaken the security of the build-pipeline? Then try looking for an alternative in which the vulnerability is fixed.

When the sources are not available, one can decompile the app and check the JAR files. When Dexguard or ProGuard are applied properly, then version information about the library is often obfuscated and therefore gone. Otherwise you can still find the information very often in the comments of the Java files of given libraries. Tools such as MobSF can help in analyzing the possible libraries packed with the application. If you can retrieve the version of the library, either via comments, or via specific methods used in certain versions, you can look them up for CVEs by hand.

If the application is a high-risk application, you will end up vetting the library manually. In that case, there are specific requirements for native code, which you can find in the chapter "Testing Code Quality" . Next to that, it is good to vet whether all best practices for software engineering are applied.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Detecting vulnerabilities of third party libraries

Rulebook

- *Analysis method of dependencies for library apps (Required)*

7.5.2.2 Licensing of the libraries used

In order to ensure that the copyright laws are not infringed, one can best check the dependencies by using a plugin which can iterate over the different libraries, such as License Gradle Plugin. This plugin can be used by taking the following steps.

In your build.gradle file add:

```
plugins {
    id "com.github.hierynomus.license-report" version"{license_plugin_version}"
}
```

Now, after the plugin is picked up, use the following commands:

```
gradle assemble
gradle downloadLicenses
```

Now a license-report will be generated, which can be used to consult the licenses used by the third party libraries. Please check the license agreements to see whether a copyright notice needs to be included into the app and whether the license type requires to open-source the code of the application.

Similar to dependency checking, there are commercial tools which are able to check the licenses as well, such as Sonatype Nexus IQ, Sourceclear, Snyk or Blackduck.

Note: If in doubt about the implications of a license model used by a third party library, then consult with a legal specialist.

When a library contains a license in which the application IP needs to be open-sourced, check if there is an alternative for the library which can be used to provide similar functionalities.

Note: In case of a hybrid app, please check the build tools used: most of them do have a license enumeration plugin to find the licenses being used.

When the sources are not available, one can decompile the app and check the JAR files. When Dexguard or ProGuard are applied properly, then version information about the library is often gone. Otherwise you can still find it very often in the comments of the Java files of given libraries. Tools such as MobSF can help in analyzing the possible libraries packed with the application. If you can retrieve the version of the library, either via comments, or via specific methods used in certain versions, you can look them up for their licenses being used by hand.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Detecting the Licenses Used by the Libraries of the Application

7.5.3 Dynamic Analysis

The dynamic analysis of this section comprises validating whether the copyrights of the licenses have been adhered to. This often means that the application should have an about or EULA section in which the copy-right statements are noted as required by the license of the third party library.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Dynamic Analysis

Rulebook

- *Verification whether the license is complied (Required)*

7.5.4 Rulebook

1. *Pay attention to the use of third-party libraries (Recommended)*
2. *Analysis method of dependencies for library apps (Required)*
3. *Verification whether the license is complied (Required)*

7.5.4.1 Pay attention to the use of third-party libraries (Recommended)

Third-party libraries have the following drawbacks and should be examined when used.

- Vulnerabilities in the library. If you use a library that contains vulnerabilities, there is a possibility that malicious or vulnerable code may be included in your application through the library. Even if the vulnerability is not discovered at this time, there is a possibility that it will be discovered in the future. In such cases, update to a version that addresses the vulnerability or refrain from using the library if an updated version is not available.
- A license included in a library. Be aware that some libraries have licenses that require you to deploy the source code of the app you use if you use that library.

Note that this problem can occur on multiple levels. If JavaScript is used in webview, JavaScript libraries may also have this issue. The same applies to plug-ins/libraries for Cordova, React-native, and Xamarin apps.

No sample code due to the rule about caution in using third-party libraries.

If this is not noted, the following may occur.

- The application contains malicious or vulnerable code that could be exploited.
- The license included in the third-party library may require you to deploy the source code of the application.

7.5.4.2 Analysis method of dependencies for library apps (Required)

Detecting vulnerabilities in third party dependencies can be done by means of the OWASP Dependency checker. This is best done by using a gradle plugin, such as dependency-check-gradle. In order to use the plugin, the following steps need to be applied: Install the plugin from the Maven central repository by adding the following script to your build.gradle:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.owasp:dependency-check-gradle:3.2.0'
    }
}

apply plugin: 'org.owasp.dependencycheck'
```

Once gradle has invoked the plugin, you can create a report by running:

```
gradle assemble
gradle dependencyCheckAnalyze --info
```

The report will be in build/reports unless otherwise configured. Use the report in order to analyze the vulnerabilities found. See remediation on what to do given the vulnerabilities found with the libraries.

Please be advised that the plugin requires to download a vulnerability feed. Consult the documentation in case issues arise with the plugin.

Alternatively there are commercial tools which might have a better coverage of the dependencies found for the libraries being used, such as Sonatype Nexus IQ, Sourceclear, Snyk or Blackduck. The actual result of using either the OWASP Dependency Checker or another tool varies on the type of (NDK related or SDK related) libraries.

Lastly, please note that for hybrid applications, one will have to check the JavaScript dependencies with RetireJS. Similarly for Xamarin, one will have to check the C# dependencies.

When a library is found to contain vulnerabilities, then the following reasoning applies:

- Is the library packaged with the application? Then check whether the library has a version in which the vulnerability is patched. If not, check whether the vulnerability actually affects the application. If that is the case or might be the case in the future, then look for an alternative which provides similar functionality, but without the vulnerabilities.
- Is the library not packaged with the application? See if there is a patched version in which the vulnerability is fixed. If this is not the case, check if the implications of the vulnerability for the build-process. Could the vulnerability impede a build or weaken the security of the build-pipeline? Then try looking for an alternative in which the vulnerability is fixed.

When the sources are not available, one can decompile the app and check the JAR files. When Dexguard or ProGuard are applied properly, then version information about the library is often obfuscated and therefore gone. Otherwise you can still find the information very often in the comments of the Java files of given libraries. Tools such as MobSF can help in analyzing the possible libraries packed with the application. If you can retrieve the version of the library, either via comments, or via specific methods used in certain versions, you can look them up for CVEs by hand.

If the application is a high-risk application, you will end up vetting the library manually. In that case, there are specific requirements for native code, which you can find in the chapter "Testing Code Quality" . Next to that, it is good to vet whether all best practices for software engineering are applied.

If this is violated, the following may occur.

- The library contains vulnerable code that can be exploited.

7.5.4.3 Verification whether the license is complied (Required)

One feature common to all major licenses is that when distributing derivative software, you must indicate the "copyright," "license statement," "disclaimer," etc. of the OSS from which you are using the software. Specifically, "what," "where," and "how" to indicate these statements **may differ depending on each license**, so it is necessary to carefully check each license for details.

For example, the MIT License requires that the "copyright notice" and "license text" appear "on every copy or substantial part of the software" as follows

```
The above copyright notice and this permission notice shall be included in all␣
↪copies or substantial portions of the Software.
```

If this is violated, the following may occur.

- The library may contain a license that requires the app' s IP to be open source.

7.6 MSTG-CODE-6

The app catches and handles possible exceptions.

7.6.1 Notes on exception handling

Exceptions occur when an application gets into an abnormal or error state. Both Java and C++ may throw exceptions. Testing exception handling is about ensuring that the app will handle an exception and transition to a safe state without exposing sensitive information via the UI or the app's logging mechanisms.

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6 and MSTG-CODE-7) Overview

7.6.2 Static Analysis

Review the source code to understand the application and identify how it handles different types of errors (IPC communications, remote services invocation, etc.). Here are some examples of things to check at this stage:

- Make sure that the application uses a well-designed and unified scheme to handle exceptions.
- Plan for standard RuntimeExceptions (e.g.NullPointerException, IndexOutOfBoundsException, ActivityNotFoundException, CancellationException, SQLException) by creating proper null checks, bound checks, and the like. An overview of the available subclasses of RuntimeException can be found in the Android developer documentation. A child of RuntimeException should be thrown intentionally, and the intent should be handled by the calling method.
- Make sure that for every non-runtime Throwable there's a proper catch handler, which ends up handling the actual exception properly.
- When an exception is thrown, make sure that the application has centralized handlers for exceptions that cause similar behavior. This can be a static class. For exceptions specific to the method, provide specific catch blocks.
- Make sure that the application doesn't expose sensitive information while handling exceptions in its UI or log-statements. Ensure that exceptions are still verbose enough to explain the issue to the user.
- Make sure that all confidential information handled by high-risk applications is always wiped during execution of the finally blocks.

```
byte[] secret;
try{
    //use secret
} catch (SPECIFICEXCEPTIONCLASS | SPECIFICEXCEPTIONCLASS2 e) {
    // handle any issues
} finally {
    //clean the secret.
}
```

Adding a general exception handler for uncaught exceptions is a best practice for resetting the application's state when a crash is imminent:

```
public class MemoryCleanerOnCrash implements Thread.UncaughtExceptionHandler {

    private static final MemoryCleanerOnCrash S_INSTANCE = new␣
↪MemoryCleanerOnCrash();
    private final List<Thread.UncaughtExceptionHandler> mHandlers = new ArrayList<>
↪();

    //initialize the handler and set it as the default exception handler
```

(continues on next page)

(continued from previous page)

```
    public static void init() {
        S_INSTANCE.mHandlers.add(Thread.getDefaultUncaughtExceptionHandler());
        Thread.setDefaultUncaughtExceptionHandler(S_INSTANCE);
    }

     //make sure that you can still add exception handlers on top of it (required␣
↪for ACRA for instance)
    public void subscribeCrashHandler(Thread.UncaughtExceptionHandler handler) {
        mHandlers.add(handler);
    }

    @Override
    public void uncaughtException(Thread thread, Throwable ex) {

            //handle the cleanup here
            //....
            //and then show a message to the user if possible given the context

        for (Thread.UncaughtExceptionHandler handler : mHandlers) {
            handler.uncaughtException(thread, ex);
        }
    }
}
```

Now the handler's initializer must be called in your custom Application class (e.g., the class that extends Application):

```
@Override
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);
    MemoryCleanerOnCrash.init();
}
```

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6 and MSTG-CODE-7) Static Analysis

Rulebook

- *Proper implementation and confirmation of exception/error processing (Required)*
- *Best practices in case of exception that cannot be catch (Recommended)*

7.6.3 Dynamic Analysis

There are several ways to do dynamic analysis:

- Use Xposed to hook into methods and either call them with unexpected values or overwrite existing variables with unexpected values (e.g., null values).
- Type unexpected values into the Android application's UI fields.
- Interact with the application using its intents, its public providers, and unexpected values.
- Tamper with the network communication and/or the files stored by the application.

The application should never crash; it should

- recover from the error or transition into a state in which it can inform the user of its inability to continue,
- if necessary, tell the user to take appropriate action (The message should not leak sensitive information.),
- not provide any information in logging mechanisms used by the application.

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6 and MSTG-CODE-7) Dynamic Analysis

7.6.4 Rulebook

1. *Proper implementation and confirmation of exception/error processing (Required)*
2. *Best practices in case of exception that cannot be catch (Recommended)*

7.6.4.1 Proper implementation and confirmation of exception/error processing (Required)

When implementing exception/error handling in Android, it is necessary to check the following.

- Make sure that the application uses a well-designed and unified scheme to handle exceptions.
- Plan for standard RuntimeExceptions (e.g.NullPointerException, IndexOutOfBoundsException, ActivityNotFoundException, CancellationException, SQLException) by creating proper null checks, bound checks, and the like. An overview of the available subclasses of RuntimeException can be found in the Android developer documentation. A child of RuntimeException should be thrown intentionally, and the intent should be handled by the calling method.
- Make sure that for every non-runtime Throwable there's a proper catch handler, which ends up handling the actual exception properly.
- When an exception is thrown, make sure that the application has centralized handlers for exceptions that cause similar behavior. This can be a static class. For exceptions specific to the method, provide specific catch blocks.
- Make sure that the application doesn't expose sensitive information while handling exceptions in its UI or log-statements. Ensure that exceptions are still verbose enough to explain the issue to the user.
- Make sure that all confidential information handled by high-risk applications is always wiped during execution of the finally blocks.

If this is violated, the following may occur.

- Application crashes.
- Confidential information is leaked.

7.6.4.2 Best practices in case of exception that cannot be catch (Recommended)

Adding a general exception handler for uncaught exceptions is a best practice for resetting the application's state when a crash is imminent:

```
public class MemoryCleanerOnCrash implements Thread.UncaughtExceptionHandler {

    private static final MemoryCleanerOnCrash S_INSTANCE = new␣
↪MemoryCleanerOnCrash();
    private final List<Thread.UncaughtExceptionHandler> mHandlers = new ArrayList<>
↪();

    //initialize the handler and set it as the default exception handler
    public static void init() {
        S_INSTANCE.mHandlers.add(Thread.getDefaultUncaughtExceptionHandler());
        Thread.setDefaultUncaughtExceptionHandler(S_INSTANCE);
    }

     //make sure that you can still add exception handlers on top of it (required␣
↪for ACRA for instance)
    public void subscribeCrashHandler(Thread.UncaughtExceptionHandler handler) {
        mHandlers.add(handler);
    }
```

(continues on next page)

(continued from previous page)

```
    @Override
    public void uncaughtException(Thread thread, Throwable ex) {

            //handle the cleanup here
            //....
            //and then show a message to the user if possible given the context

        for (Thread.UncaughtExceptionHandler handler : mHandlers) {
            handler.uncaughtException(thread, ex);
        }
    }
}
```

Now the handler's initializer must be called in your custom Application class (e.g., the class that extends Application):

```
@Override
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);
    MemoryCleanerOnCrash.init();
}
```

If this is violated, the following may occur.

- Application crashes.
- Confidential information is leaked.

7.7 MSTG-CODE-7

Error handling logic in security controls denies access by default.

* Since the descriptions related to exception handling are summarized in "7.6.1. Notes on exception handling", descriptions in this chapter are omitted.

7.8 MSTG-CODE-8

In unmanaged code, memory is allocated, freed and used securely.

7.8.1 Memory Corruption Bugs in native code

Memory corruption bugs are a popular mainstay with hackers. This class of bug results from a programming error that causes the program to access an unintended memory location. Under the right conditions, attackers can capitalize on this behavior to hijack the execution flow of the vulnerable program and execute arbitrary code. This kind of vulnerability occurs in a number of ways:

The primary goal in exploiting memory corruption is usually to redirect program flow into a location where the attacker has placed assembled machine instructions referred to as shellcode. On iOS, the data execution prevention feature (as the name implies) prevents execution from memory defined as data segments. To bypass this protection, attackers leverage return-oriented programming (ROP). This process involves chaining together small, pre-existing code chunks ("gadgets") in the text segment where these gadgets may execute a function useful to the attacker or, call mprotect to change memory protection settings for the location where the attacker stored the shellcode.

Android apps are, for the most part, implemented in Java which is inherently safe from memory corruption issues by design. However, native apps utilizing JNI libraries are susceptible to this kind of bug.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.1 Buffer overflows

This describes a programming error where an app writes beyond an allocated memory range for a particular operation. An attacker can use this flaw to overwrite important control data located in adjacent memory, such as function pointers. Buffer overflows were formerly the most common type of memory corruption flaw, but have become less prevalent over the years due to a number of factors. Notably, awareness among developers of the risks in using unsafe C library functions is now a common best practice plus, catching buffer overflow bugs is relatively simple. However, it is still worth testing for such defects.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.2 Out-of-bounds-access

Buggy pointer arithmetic may cause a pointer or index to reference a position beyond the bounds of the intended memory structure (e.g. buffer or list). When an app attempts to write to an out-of-bounds address, a crash or unintended behavior occurs. If the attacker can control the target offset and manipulate the content written to some extent, code execution exploit is likely possible.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.3 Dangling pointers

These occur when an object with an incoming reference to a memory location is deleted or deallocated, but the object pointer is not reset. If the program later uses the dangling pointer to call a virtual function of the already deallocated object, it is possible to hijack execution by overwriting the original vtable pointer. Alternatively, it is possible to read or write object variables or other memory structures referenced by a dangling pointer.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.4 Use-after-free

This refers to a special case of dangling pointers referencing released (deallocated) memory. After a memory address is cleared, all pointers referencing the location become invalid, causing the memory manager to return the address to a pool of available memory. When this memory location is eventually re-allocated, accessing the original pointer will read or write the data contained in the newly allocated memory. This usually leads to data corruption and undefined behavior, but crafty attackers can set up the appropriate memory locations to leverage control of the instruction pointer.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.5 Integer overflows

When the result of an arithmetic operation exceeds the maximum value for the integer type defined by the programmer, this results in the value "wrapping around" the maximum integer value, inevitably resulting in a small value being stored. Conversely, when the result of an arithmetic operation is smaller than the minimum value of the integer type, an integer underflow occurs where the result is larger than expected. Whether a particular integer overflow/underflow bug is exploitable depends on how the integer is used. For example, if the integer type were to represent the length of a buffer, this could create a buffer overflow vulnerability.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.6 Format string vulnerabilities

When unchecked user input is passed to the format string parameter of the printf family of C functions, attackers may inject format tokens such as '%c' and '%n' to access memory. Format string bugs are convenient to exploit due to their flexibility. Should a program output the result of the string formatting operation, the attacker can read and write to memory arbitrarily, thus bypassing protection features such as ASLR.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.2 Buffer and Integer Overflows

The following code snippet shows a simple example for a condition resulting in a buffer overflow vulnerability.

```
void copyData(char *userId) {
   char  smallBuffer[10]; // size of 10
   strcpy(smallBuffer, userId);
}
```

To identify potential buffer overflows, look for uses of unsafe string functions (strcpy, strcat, other functions beginning with the "str" prefix, etc.) and potentially vulnerable programming constructs, such as copying user input into a limited-size buffer. The following should be considered red flags for unsafe string functions:

- strcat
- strcpy
- strncat
- strlcat
- strncpy
- strlcpy
- sprintf
- snprintf
- gets

Also, look for instances of copy operations implemented as "for" or "while" loops and verify length checks are performed correctly.

Verify that the following best practices have been followed:

- When using integer variables for array indexing, buffer length calculations, or any other security-critical operation, verify that unsigned integer types are used and perform precondition tests are performed to prevent the possibility of integer wrapping.
- The app does not use unsafe string functions such as strcpy, most other functions beginning with the "str" prefix, sprint, vsprintf, gets, etc.;
- If the app contains C++ code, ANSI C++ string classes are used;
- In case of memcpy, make sure you check that the target buffer is at least of equal size as the source and that both buffers are not overlapping.
- No untrusted data is concatenated into format strings.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Buffer and Integer Overflows

Rulebook

- *Do not use non-safe character string functions that cause buffer overflow (Required)*

- *Buffer overflow best practices (Required)*

7.8.2.1 Static Analysis

Static code analysis of low-level code is a complex topic that could easily fill its own book. Automated tools such as RATS combined with limited manual inspection efforts are usually sufficient to identify low-hanging fruits. However, memory corruption conditions often stem from complex causes. For example, a use-after-free bug may actually be the result of an intricate, counter-intuitive race condition not immediately apparent. Bugs manifesting from deep instances of overlooked code deficiencies are generally discovered through dynamic analysis or by testers who invest time to gain a deep understanding of the program.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Static Analysis

7.8.2.2 Dynamic Analysis

Memory corruption bugs are best discovered via input fuzzing: an automated black-box software testing technique in which malformed data is continually sent to an app to survey for potential vulnerability conditions. During this process, the application is monitored for malfunctions and crashes. Should a crash occur, the hope (at least for security testers) is that the conditions creating the crash reveal an exploitable security flaw.

Fuzz testing techniques or scripts (often called "fuzzers") will typically generate multiple instances of structured input in a semi-correct fashion. Essentially, the values or arguments generated are at least partially accepted by the target application, yet also contain invalid elements, potentially triggering input processing flaws and unexpected program behaviors. A good fuzzer exposes a substantial amount of possible program execution paths (i.e. high coverage output). Inputs are either generated from scratch ("generation-based") or derived from mutating known, valid input data ("mutation-based").

For more information on fuzzing, refer to the OWASP Fuzzing Guide.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Dynamic Analysis

7.8.3 Memory Corruption Bug in Java/Kotlin code

Android applications often run on a VM where most of the memory corruption issues have been taken care off. This does not mean that there are no memory corruption bugs. Take CVE-2018-9522 for instance, which is related to serialization issues using Parcels. Next, in native code, we still see the same issues as we explained in the general memory corruption section. Last, we see memory bugs in supporting services, such as with the Stagefright attack as shown at BlackHat.

A memory leak is often an issue as well. This can happen for instance when a reference to the Context object is passed around to non-Activity classes, or when you pass references to Activity classes to your helper classes.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.3.1 Static Analysis

There are various items to look for:

- Are there native code parts? If so: check for the given issues in the general memory corruption section. Native code can easily be spotted given JNI-wrappers, .CPP/.H/.C files, NDK or other native frameworks.
- Is there Java code or Kotlin code? Look for Serialization/deserialization issues, such as described in A brief history of Android deserialization vulnerabilities.

Note that there can be Memory leaks in Java/Kotlin code as well. Look for various items, such as: BroadcastReceivers which are not unregistered, static references to Activity or View classes, Singleton classes that have references to

Context, Inner Class references, Anonymous Class references, AsyncTask references, Handler references, Threading done wrong, TimerTask references. For more details, please check:

- 9 ways to avoid memory leaks in Android
- Memory Leak Patterns in Android.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Static Analysis

Rulebook

- *Serilizer/decoritalization problem (Recommended)*
- *Search for items that may have a memory leak (Recommended)*

7.8.3.2 Dynamic Analysis

There are various steps to take:

- In case of native code: use Valgrind or Mempatrol to analyze the memory usage and memory calls made by the code.
- In case of Java/Kotlin code, try to recompile the app and use it with Squares leak canary.
- Check with the Memory Profiler from Android Studio for leakage.
- Check with the Android Java Deserialization Vulnerability Tester, for serialization vulnerabilities.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Dynamic Analysis

7.8.4 Rulebook

1. *Do not use non-safe character string functions that cause buffer overflow (Required)*
2. *Buffer overflow best practices (Required)*
3. *Serilizer/decoritalization problem (Recommended)*
4. *Search for items that may have a memory leak (Recommended)*

7.8.4.1 Do not use non-safe character string functions that cause buffer overflow (Required)

The following unsafe string functions exist that cause buffer overflows and should be avoided before they occur.

- strcat
- strcpy
- strncat
- strlcat
- strncpy
- strlcpy
- sprintf
- snprintf
- gets

* No sample code due to deprecated rules.

If this is violated, the following may occur.

- May cause buffer overflow.

7.8.4.2 Buffer overflow best practices (Required)

To avoid buffer overflows, check the following

- When using integer variables for array indexing, buffer length calculations, or any other security-critical operation, verify that unsigned integer types are used and perform precondition tests are performed to prevent the possibility of integer wrapping.
- The app does not use unsafe string functions such as strcpy, most other functions beginning with the "str" prefix, sprint, vsprintf, gets, etc.;
- If the app contains C++ code, ANSI C++ string classes are used;
- In case of memcpy, make sure you check that the target buffer is at least of equal size as the source and that both buffers are not overlapping.
- iOS apps written in Objective-C use NSString class. C apps on iOS should use CFString, the Core Foundation representation of a string.
- No untrusted data is concatenated into format strings.

If this is violated, the following may occur.

- May cause buffer overflow.

7.8.4.3 Serilizer/decoritalization problem (Recommended)

As explained in A brief history of Android deserialization vulnerabilities. Problems. We identified a number of vulnerabilities related to Android deserialization and showed how a flaw in Android' s IPC mechanism can lead to four different exploitable vulnerabilities.

- CVE-2014-7911: Privilege Escalation using ObjectInputStream
- Finding C++ proxy classes with CodeQL
- CVE-2015-3825: One class to rule them all
- CVE-2017-411 and CVE-2017-412: Ashmem race conditions in MemoryIntArray

If this is not noted, the following may occur.

- Attacks due to deserialization vulnerabilities.

7.8.4.4 Search for items that may have a memory leak (Recommended)

- Unregistered BroadcastReceiver
- Static reference to Activity or View class
- Singleton class with a reference to Context
- Reference to Inner class
- Reference to Anonymous class
- Reference to AsyncTask
- References to Handler
- Threading errors
- See TimerTask

For more details, please check:

- 9 ways to avoid memory leaks in Android
- Memory Leak Patterns in Android.

If this is not noted, the following may occur.

- May cause memory leaks.

7.9 MSTG-CODE-9

Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated.

7.9.1 Use of Binary Protection Mechanisms

The tests used to detect the presence of binary protection mechanisms heavily depend on the language used for developing the application.

In general all binaries should be tested, which includes both the main app executable as well as all libraries/dependencies. However, on Android we will focus on native libraries since the main executables are considered safe as we will see next.

Android optimizes its Dalvik bytecode from the app DEX files (e.g. classes.dex) and generates a new file containing the native code, usually with an .odex, .oat extension. This Android compiled binary is wrapped using the ELF format which is the format used by Linux and Android to package assembly code.

The app's NDK native libraries also use the ELF format.

- PIE (Position Independent Executable):
  - Since Android 7.0 (API level 24), PIC compilation is enabled by default for the main executables.
  - With Android 5.0 (API level 21), support for non-PIE enabled native libraries was dropped and since then, PIE is enforced by the linker.
- Memory management:
  - Garbage Collection will simply run for the main binaries and there's nothing to be checked on the binaries themselves.
  - Garbage Collection does not apply to Android native libraries. The developer is responsible for doing proper *Manual Memory Management*. See "*Memory Corruption Bugs in native code*".
- Stack Smashing Protection:
  - Android apps get compiled to Dalvik bytecode which is considered memory safe (at least for mitigating buffer overflows). Other frameworks such as Flutter will not compile using stack canaries because of the way their language, in this case Dart, mitigates buffer overflows.
  - It must be enabled for Android native libraries but it might be difficult to fully determine it.
    - NDK libraries should have it enabled since the compiler does it by default.
    - Other custom C/C++ libraries might not have it enabled.

Learn more:

- Android executable formats
- Android runtime (ART)
- Android NDK
- Android linker changes for NDK developers

Reference

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Overview

7.9.1.1 PIC (Position Independent Code)

PIC (Position Independent Code) is code that, being placed somewhere in the primary memory, executes properly regardless of its absolute address. PIC is commonly used for shared libraries, so that the same library code can be loaded in a location in each program address space where it does not overlap with other memory in use (for example, other shared libraries).

Reference

- owasp-mastg Binary Protection Mechanisms Position Independent Code

7.9.1.2 PIE (Position Independent Executable)

PIE (Position Independent Executable) are executable binaries made entirely from PIC. PIE binaries are used to enable ASLR (Address Space Layout Randomization) which randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

Reference

- owasp-mastg Binary Protection Mechanisms Position Independent Code

7.9.1.3 Memory management

Automatic Reference Counting

ARC (Automatic Reference Counting) is a memory management feature of the Clang compiler exclusive to Objective-C and Swift. ARC automatically frees up the memory used by class instances when those instances are no longer needed. ARC differs from tracing garbage collection in that there is no background process that deallocates the objects asynchronously at runtime.

Unlike tracing garbage collection, ARC does not handle reference cycles automatically. This means that as long as there are "strong" references to an object, it will not be deallocated. Strong cross-references can accordingly create deadlocks and memory leaks. It is up to the developer to break cycles by using weak references. You can learn more about how it differs from Garbage Collection here.

Reference

- owasp-mastg Binary Protection Mechanisms Automatic Reference Counting

Garbage Collection

Garbage Collection (GC) is an automatic memory management feature of some languages such as Java/Kotlin/Dart. The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced —also called garbage. The Android runtime (ART) makes use of an improved version of GC. You can learn more about how it differs from ARC here.

Reference

- owasp-mastg Binary Protection Mechanisms Garbage Collection

Manual Memory Management

Manual memory management is typically required in native libraries written in C/C++ where ARC and GC do not apply. The developer is responsible for doing proper memory management. Manual memory management is known to enable several major classes of bugs into a program when used incorrectly, notably violations of memory safety or memory leaks.

More information can be found in "*Memory Corruption Bugs in native code*" .

Reference

- owasp-mastg Binary Protection Mechanisms Manual Memory Management

7.9.1.4 Stack Smashing Protection

Stack canaries help prevent stack buffer overflow attacks by storing a hidden integer value on the stack right before the return pointer. This value is then validated before the return statement of the function is executed. A buffer overflow attack often overwrites a region of memory in order to overwrite the return pointer and take over the program flow. If stack canaries are enabled, they will be overwritten as well and the CPU will know that the memory has been tampered with.

Stack buffer overflow is a type of the more general programming vulnerability known as buffer overflow (or buffer overrun). Overfilling a buffer on the stack is more likely to derail program execution than overfilling a buffer on the heap because the stack contains the return addresses for all active function calls.

Reference

- owasp-mastg Binary Protection Mechanisms Stack Smashing Protection

7.9.2 Static Analysis

Test the app native libraries to determine if they have the PIE and stack smashing protections enabled.

You can use radare2's rabin2 to get the binary information. We'll use the UnCrackable App for Android Level 4 v1.0 APK as an example.

All native libraries must have canary and pic both set to true.

That's the case for libnative-lib.so:

```
rabin2 -I lib/x86_64/libnative-lib.so | grep -E "canary|pic"
canary   true
pic      true
```

But not for libtool-checker.so:

```
rabin2 -I lib/x86_64/libtool-checker.so | grep -E "canary|pic"
canary   false
pic      true
```

In this example, libtool-checker.so must be recompiled with stack smashing protection support.

Reference

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Static Analysis

Rulebook

- *Make sure the protection of PIE and stack smashing is enabled (Required)*

7.9.3 Rulebook

1. *Make sure the protection of PIE and stack smashing is enabled (Required)*

7.9.3.1 Make sure the protection of PIE and stack smashing is enabled (Required)

Test the app native libraries to determine if they have the PIE and stack smashing protections enabled.

You can use radare2's rabin2 to get the binary information. We'll use the UnCrackable App for Android Level 4 v1.0 APK as an example.

All native libraries must have canary and pic both set to true.

That's the case for libnative-lib.so:

```
rabin2 -I lib/x86_64/libnative-lib.so | grep -E "canary|pic"
canary   true
pic      true
```

But not for libtool-checker.so:

```
rabin2 -I lib/x86_64/libtool-checker.so | grep -E "canary|pic"
canary   false
pic      true
```

In this example, libtool-checker.so must be recompiled with stack smashing protection support.

If this is violated, the following may occur.

- May cause stack buffer overflow.

8 Revision history

2023-04-01

Initial English Edition