MASDG

モバイルアプリケーション セキュリティ設計ガイド Android 版

OWASP®

目次

2023 年 04 月 01 日版

株式会社ラック

モバイルアプリケーションのセキュリティ設計ガイド（MASDG）Android 版へようこそ。

モバイルアプリケーションのセキュリティ設計ガイド Android 版は OWASP が公開している MASVS および MASTG に当社独自の判定基準（ルールブック）やサンプルコードを組み込みんだ Android 上のセキュアなモバイルアプリケーションを設計、開発、テストするときに必要となるセキュリティ設計のフレームワークを確立するためのドキュメントです。

MASDG は、セキュリティ要件に対する具体的な設計内容に特化したベストプラクティスやサンプル等を取り扱うことで、MASVS を元に検討したセキュリティ要件からセキュリティ設計を作成することをサポートすることや、MASTG で示されているテスト方法を実施する前にセキュリティ設計に問題が無いかを評価することをサポートします。

当社の独自のルールブックは MASVS L1 検証標準を対象としておりモバイルアプリを開発する際に網羅的なセキュリティベースラインを提供することを目的としています。これから創出される新しいテクノロジーは必ずリスクをもたらしプライバシーやセーフティーの課題を生じますがモバイルアプリに関する脅威に立ち向かうべく本ドキュメントの作成を行いました。

MASVS および MASTG に対しては様々なコミュニティや業界からフィードバックを受けていますが、当社としても社会生活に不可欠となったモバイルアプリケーションに対するセキュリティリスクの課題に取り組んで行きたいと考えておりますので MASDG を開発し公開しました。皆様からのフィードバックを歓迎します。

著作権とライセンス

- 本ガイドの内容は執筆時点のものです。サンプルコードを使用する場合はこの点にあらかじめご注意ください。
- 株式会社ラックならびに執筆関係者は、このガイド文書に関するいかなる責任も負うものではありません。全ては自己責任にてご活用ください。
- Android は、Google, LLC. の商標または登録商標です。また、本文書に登場する会社名、製品名、サービス名は、一般に各社の登録商標または商標です。本文中では ®、TM、© マークは明記していません。
- この文書の内容の一部は、OWASP MASVS, OWASP MASTG が作成、提供しているコンテンツをベースに複製、改版したものです。

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

1 アーキテクチャ・設計・脅威モデリング要件

1.1 MSTG-ARCH-1

アプリのすべてのコンポーネントを把握し、それらが必要とされている。

1.1.1 コンポーネント

アプリのすべてのコンポーネントを把握して、不要なコンポーネントを削除すること。

主なコンポーネントの種類は以下の通り。

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

1.2 MSTG-ARCH-2

セキュリティコントロールはクライアント側だけではなくそれぞれのリモートエンドポイントで実施されている。

1.2.1 認証・認可情報の改竄

1.2.1.1 適切な認証対応

認証・認可のテストを行う場合は、以下の手順で行う。

- アプリが使用する追加の認証要素を特定する。
- 重要な機能を提供するすべてのエンドポイントを特定する。
- すべてのサーバ側エンドポイントにおいて、追加要素が厳密に実施されていることを確認する。

認証バイパスの脆弱性は、サーバ上で認証状態が一貫して実施されておらず、クライアントが認証状態を改ざんできる場合に存在する。バックエンドサービスは、モバイルクライアントからのリクエストを処理している間、リソースがリクエストされるたびに、ユーザがログインしているか、認可されているかを確認し、一貫した認可チェックを実施する必要がある。

OWASP Web Testing Guide の次の例を検討してください。この例では、Web リソースは URL を介してアクセスされ、認証状態は GET パラメータを介して渡される。

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

クライアントは、リクエストと共に送られる GET パラメータを任意に変更することができる。クライアントが単に authenticated パラメータの値を "yes" に変更することを妨げるものはなにもなく、効果的に認証をバイパスすることができてしまう。

これはおそらく実際には見られない単純な例だが、プログラマは認証状態を維持するために Cookie のような「隠れた」クライアント側パラメータに依存することがある。このようなパラメータは改ざんできないと想定している。例えば、Nortel Contact Center Manager における次のような典型的な脆弱性が考えられる。Nortel のアプライアンスの管理 Web アプリケーションは、cookie の「isAdmin」に依存して、ログインしたユーザに管理権限を付与する必要があるかどうかを判断していた。その結果、以下のようにクッキーの値を設定するだけで、管理者権限を取得することが可能であった。

```
isAdmin=True
```

セキュリティの専門家は、以前はセッションベースの認証を使用し、サーバ上でのみセッションデータを維持することを推奨していた。これにより、クライアント側でセッションの状態が改ざんされることを防ぐことができる。しかし、セッションベースの認証の代わりにステートレス認証を使用する要点は、サーバ上にセッションの状態を持たないことである。代わり、状態はクライアント側のトークンに保存され、リクエストごとに送信される。この場合、isAdmin のようなクライアント側のパラメータを見ることは、全く問題ない。

改ざんを防ぐために、クライアント側のトークンには暗号署名が付加されている。もちろん、うまくいかないこともあり、一般的なステートレス認証の実装は攻撃に対して脆弱である。例えば、いくつかの JSON Web Token(JWT) 実装の署名検証は、署名タイプを「None」に設定することで無効化することが可能である。この攻撃については、「Testing JSON Web Token」の章で詳しく説明する。

参考資料

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

1.2.2 インジェクション欠陥

インジェクションの欠陥は、ユーザ入力がバックエンドのクエリやコマンドに挿入されたときに発生するセキュリティ上の脆弱性のことである。メタ文字を挿入することで、攻撃者は、コマンドやクエリの一部として誤って解釈される悪意のあるコードを実行できる。例えば、SQL クエリを操作することで、攻撃者は任意のデータベースレコードを取得したり、バックエンドのデータベースの内容を操作したりすることができる。

このクラスの脆弱性は、サーバ側の Web サービスに最も多く存在する。モバイルアプリにも悪用可能な例があるが、発生頻度は低く、攻撃対象領域も小さくなっている。

例えば、アプリがローカルの SQLite データベースにクエリを実行する場合、そのようなデータベースは通常、機密データを保存しない（開発者が基本的なセキュリティプラクティスに従っている場合）。このため、SQL インジェクションは攻撃ベクトルとしてふさわしくない。それにもかかわらず、インジェクションの脆弱性が悪用されることもあるため、適切な入力検証はプログラマにとって必要なベストプラクティスであると言える。

参考資料

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

ルールブック

- 適切な入力検証を実施する（必須）

1.2.2.1 SQL インジェクション

SQL インジェクション攻撃は、あらかじめ定義された SQL コマンドの構文を模倣して、入力データに SQL コマンドを統合する。SQL インジェクション攻撃が成功すると、攻撃者は、サーバから付与された権限に応じて、データベースへの読み取りまたは書き込みが可能になり、管理コマンドを実行できる可能性がある。

Android と iOS のアプリは、ローカルデータストレージを制御および整理する手段として SQLite データベースを使用する。例えば、Android アプリが、ローカルデータベースにユーザ情報を保存して、ローカルユーザ認証を処理しているとする（例のために見過ごしている不適切なプログラミング手法である）。ログイン時に、アプリはデータベースにクエリを実行し、ユーザが入力したユーザ名とパスワードを使用してレコードを検索する。

```
SQLiteDatabase db;

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

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

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

さらに、攻撃者が「ユーザ名」と「パスワード」の欄に以下の値を入力したとする。

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

これにより、以下のようなクエリが発生する。

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

条件 '1' = '1' は常に true と評価されるため、このクエリはデータベース内のすべてのレコードを返し、有効なユーザ アカウントが入力されていなくても、ログイン関数は true を返す。Ostorlab は、この SQL インジェクションのペイロードを使用して、adb を使用して Yahoo's weather mobile application のソートパラメータを悪用した。

Mark Woods 氏は、QNAP NAS ストレージ・アプライアンス上で動作する「Qnotes」および「Qget」Android アプリ内で、クライアント側 SQL インジェクションのもう 1 つの実例を発見した。これらのアプリは、SQL インジェクションに対して脆弱なコンテンツプロバイダをエクスポートし、攻撃者が NAS デバイスの認証情報を取得することを可能にする。この問題の詳細な説明は、Nettitude Blog にある。

参考資料

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

1.2.2.2 XML インジェクション

XML インジェクション攻撃では、攻撃者は XML メタ文字を挿入して XML コンテンツを構造的に変更する。これは、XML ベースのアプリケーションまたはサービスのロジックを危険にさらすだけでなく、コンテンツを処理する XML パーサーの操作を攻撃者が悪用できる可能性もある。

この攻撃の一般的な変種として、XML eXternal Entity (XXE) がある。攻撃者は、URI を含む外部エンティティ定義を入力 XML に挿入する。XML パーサーは、解析中に URI で指定されたリソースにアクセスし、攻撃者が定義したエンティティを展開する。

解析アプリケーションの完全性は、最終的に攻撃者の与えられる機能を決定し、悪意のあるユーザは次のいずれか（あるいはすべて）を行うことができる。ローカルファイルへのアクセス、任意のホストとポートへの HTTP リクエストの誘因、クロスサイトリクエストフォージェリ（CSRF）攻撃の起動、サービス拒否状態の誘因。OWASP ウェブテストガイドには、XXE に関する以下の例がある。

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

この例では、ローカルファイル /dev/random が開かれ、そこで無限のバイトストリームが返されるため、サービス拒否が引き起こされる可能性がある。

現在のアプリ開発のトレンドは、XML が一般的でなくなってきているため、ほとんどが REST/JSON ベースのサービスに焦点を当てている。しかし、まれにユーザが提供したコンテンツや信頼できないコンテンツを使用して XML クエリを作成した場合、iOS の NSXMLParser などのローカル XML パーサーによって解釈される可能性がある。そのため、入力は常に検証する必要があり、メタ文字はエスケープする必要がある。

参考資料

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

ルールブック

- 入力は常に検証する必要があり、メタ文字はエスケープする必要がある（必須）

1.2.2.3 インジェクション攻撃ベクトル

モバイルアプリの攻撃対象は、一般的な Web アプリケーションやネットワークアプリケーションとは全く異なる。モバイルアプリは、ネットワーク上にサービスを公開することがあまりなく、アプリのユーザインターフェース上で実行可能な攻撃ベクトルは稀である。アプリに対するインジェクション攻撃は、プロセス間通信（IPC）インターフェースを介して発生する可能性が最も高く、悪意のあるアプリがデバイス上で実行されている別のアプリを攻撃することがある。

潜在的な脆弱性を見つけるには、まず次のどちらかを行う。

- 信頼できない入力の可能性のあるエントリポイントを特定し、その場所からトレースして、宛先に潜在的な脆弱性のある関数が含まれているかどうかを確認する。
- 既知の危険なライブラリ/API 呼び出し (SQL クエリなど) を特定し、未チェックの入力がそれぞれのクエリと正常に連携しているかどうかを確認する。

手動のセキュリティ レビューでは、両方の手法を組み合わせて使用する必要がある。一般に、信頼できない入力は、次のチャネルを通じてモバイル アプリに侵入する。

- IPC コール
- カスタム URL スキーム
- QR コード
- Bluetooth、NFC などで受信した入力データ
- ペーストボード
- ユーザインターフェース

以下のベストプラクティスに従っていることを確認する。

- 信頼できない入力は、許容値のリストを使用して型チェックや検証を行う。
- データベースクエリを実行するときは、変数バインディング (つまり、パラメータ化されたクエリ) を使用する prepared ステートメントが使用される。prepared ステートメントが定義されている場合、ユーザ提供のデータと SQL コードは自動的に分離される。
- XML データを解析するときは、XXE 攻撃を防ぐために、パーサーアプリケーションが外部エンティティの解決を拒否するように構成されていることを確認する。
- X.509 形式の証明書データを扱う場合は、安全なパーサーが使用されていることを確認する。例えば、バージョン 1.6 未満の Bouncy Castle では、安全でないリフレクションによるリモートコード実行が可能である。

各モバイル OS の入力ソースや脆弱性の可能性のある API に関する詳細は、OS 固有のテストガイドで参照。

参考資料

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

ルールブック

- 宛先に潜在的な脆弱性のある関数を含めない（必須）
- 未チェックの入力がそれぞれのクエリと正常に連携している（必須）
- 信頼できない入力を確認する（必須）
- パーサーアプリケーションは外部エンティティの解決を拒否するように構成する（必須）
- *X.509* 形式の証明書データを利用する場合は安全なパーサーを使用する（必須）

1.2.3 ルールブック

1. 適切な入力検証を実施する（必須）
2. 入力は常に検証する必要があり、メタ文字はエスケープする必要がある（必須）
3. 宛先に潜在的な脆弱性のある関数を含めない（必須）
4. 未チェックの入力がそれぞれのクエリと正常に連携している（必須）
5. 信頼できない入力を確認する（必須）
6. パーサーアプリケーションは外部エンティティの解決を拒否するように構成する（必須）
7. *X.509* 形式の証明書データを利用する場合は安全なパーサーを使用する（必須）

1.2.3.1 適切な入力検証を実施する（必須）

インジェクションの脆弱性が悪用されることもあるため、適切な入力検証はプログラマにとって必要なベストプラクティスであると言える。

下記は入力検証の一例。

- 正規表現チェック
- 長さ/サイズチェック

これに違反する場合、以下の可能性がある。

- インジェクションの脆弱性が悪用される可能性がある。

1.2.3.2 入力は常に検証する必要があり、メタ文字はエスケープする必要がある（必須）

ユーザが提供したコンテンツや信頼できないコンテンツを使用して XML クエリを作成した場合、ローカル XML パーサーによって XML メタ文字が XML コンテンツとして解釈される可能性がある。そのため、入力は常に検証する必要があり、メタ文字はエスケープする必要がある。

下記は入力検証の一例。

- 正規表現チェック
- 長さ/サイズチェック

下記はメタ文字の一例。

表 1.2.3.2.1　メタ文字一覧

文字	項目名	エンティティ参照による表記
<	右大なり	&lt;
>	左大なり	&gt;
&	アンパーサント	&amp;
"	ダブルクォーテーション	&quot;
'	シングルクォーテーション	&apos;

これに違反する場合、以下の可能性がある。

- ローカル XML パーサーによって XML メタ文字が XML コンテンツとして解釈される可能性がある。

1.2.3.3 宛先に潜在的な脆弱性のある関数を含めない（必須）

信頼できない入力の可能性のあるエントリポイントを特定し、その場所からトレースして、宛先に潜在的な脆弱性のある関数（サードパーティ製の関数）が含まれているかどうかを確認する。

これに違反する場合、以下の可能性がある。

- プロセス間通信（ IPC ）インターフェースを介して、悪意のあるアプリがデバイス上で実行されている別のアプリを攻撃する可能性がある。

1.2.3.4 未チェックの入力がそれぞれのクエリと正常に連携している（必須）

既知の危険なライブラリ /API 呼び出し (SQL クエリなど) を特定し、未チェックの入力がそれぞれのクエリと正常に連携しているかどうかを確認する。また、利用するライブラリ /API のリファレンスを確認し、非推奨でないことを確認する。

Android API リファレンス：https://developer.android.com/?hl=ja

これに違反する場合、以下の可能性がある。

- プロセス間通信（ IPC ）インターフェースを介して、悪意のあるアプリがデバイス上で実行されている別のアプリを攻撃する可能性がある。

1.2.3.5 信頼できない入力を確認する（必須）

一般に、信頼できない入力は、次のチャネルを通じてモバイルアプリに侵入するため、チャネルの利用箇所をソースコードから特定し確認する。

下記はチャネルの利用を特定するキーワードの一例。

- IPC コール：ContentProvider
- カスタム URL スキーム：scheme
- QR コード：qr, camera
- Bluetooth、NFC などで受信した入力データ：bluetoothAdapter
- ペーストボード：ClipboardManager
- ユーザインターフェース：EditText

これに違反する場合、以下の可能性がある。

- プロセス間通信（ IPC ）インターフェースを介して、悪意のあるアプリがデバイス上で実行されている別のアプリを攻撃する可能性がある。

1.2.3.6 パーサーアプリケーションは外部エンティティの解決を拒否するように構成する（必須）

XXE 攻撃を防ぐために、パーサーアプリケーションは外部エンティティの解決を拒否するように構成する必要がある。

XXE を防止する最も安全な方法は、常に DTD (外部エンティティ) を完全に無効にすることである。パーサーに応じて、メソッドは次のようになる。

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

DTD を無効にすると、パーサーは Billion Laughs などのサービス拒否 (DOS) 攻撃に対しても安全になる。DTD を完全に無効にすることができない場合は、各パーサーに固有の方法で、外部エンティティと外部ドキュメントタイプの宣言を無効にする必要がある。

これに違反する場合、以下の可能性がある。

- XXE 攻撃に対して脆弱となる。

参考資料

- OWASP Cheat Sheet Series XML External Entity Prevention

1.2.3.7 X.509 形式の証明書データを利用する場合は安全なパーサーを使用する（必須）

X.509 形式の証明書データを扱う場合は、安全なパーサーを使用する。

下記は安全なパーサーの一例。

- Bouncy Castle（バージョン 1.6 以上）

これに違反する場合、以下の可能性がある。

- 安全でないリフレクションによるリモートコード実行等、インジェクション攻撃に対して脆弱となる。

1.3 MSTG-ARCH-3

モバイルアプリと接続されるすべてのリモートサービスの高次のアーキテクチャが定義され、そのアーキテクチャにセキュリティが対応されている。

※リモートサービス側での対応に関する章であるため、本資料ではガイド記載を省略

1.4 MSTG-ARCH-4

モバイルアプリのコンテキストで機密とみなされるデータが明確に特定されている。

機密情報の分類は、業界や国によって異なる。さらに、組織は機密データに対して制限的な見方をしている場合があり、機密情報を明確に定義するデータ分類ポリシーを持っている場合がある。データ分類ポリシーが利用できない場合は、一般的に機密と見なされる情報の次のリストを使用する。

参考資料

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

ルールブック

- データ分類ポリシーに従い、機密データを識別する（必須）

1.4.1 ユーザ認証情報

ユーザ認証情報 (資格情報、PIN など)。

1.4.2 個人識別情報

個人情報の盗難に悪用される可能性のある個人識別情報 (PII): 社会保障番号、クレジット カード番号、銀行口座番号、健康情報。

1.4.3 デバイス識別子

個人を特定できるデバイス識別子。

1.4.4 機密性高いデータ

侵害されると風評被害や金銭的コストにつながる機密性の高いデータ。

1.4.5 保護が法的義務であるデータ

保護が法的義務であるデータ。

1.4.6 技術データ

アプリ (またはその関連システム) によって生成され、他のデータまたはシステム自体を保護するために使用される技術データ (暗号化キーなど)。

1.4.7 ルールブック

1. データ分類ポリシーに従い、機密データを識別する（必須）

1.4.7.1 データ分類ポリシーに従い、機密データを識別する（必須）

業界・国・組織で明確に定義されているデータ分類ポリシーに従い、機密データを識別する。データ分類ポリシーが利用できない場合は、一般的に機密と見なされる情報として以下のリストを使用する。

- ユーザ認証情報（資格情報、PIN など）
- 個人情報の盗難に悪用される可能性のある個人識別情報（ PII）：社会保障番号、クレジット カード番号、銀行口座番号、健康情報。
- 個人を特定できるデバイス識別子
- 機密性高いデータ：侵害されると風評被害や金銭的コストにつながる機密性の高いデータ。
- 保護が法的義務であるデータ
- アプリ（またはその関連システム）によって生成され、他のデータまたはシステム自体を保護するために使用される技術データ（暗号化キーなど）。

「機密データ」の定義なしで機密データの漏洩を検出することは不可能な場合があるため、定義はテストを開始する前に決定する必要がある。

これに違反する場合、以下の可能性がある。

- ペネトレーションテストの結果から侵害される可能性のある機密データを機密データとして認識できず、リスクとして把握・対策出来ない可能性がある。

1.5 MSTG-ARCH-12

アプリはプライバシーに関する法律および規制に準拠している。

1.5.1 プライバシーに関する一般的な法律および規則

参考資料

- V6.1 Data Classification
- V6.1 データ分類

1.5.1.1 個人情報・プライバシー

個人情報を扱っている場合、GDPR, 個人情報保護法に準拠していること。

1.5.1.2 医療データ

医療データを扱っている場合、HIPAA,HITECH に準拠していること。

1.5.1.3 金融情報

クレジットカード情報

クレジットカード情報を扱っている場合、PCI DSS に準拠していること。

1.5.2 プライバシーに関する Google Play での規則

Google は、ユーザのプライバシーを保護し、安全な環境をユーザに提供するように努めている。虚偽のあるアプリ、悪意のあるアプリ、ネットワーク、デバイス、個人データを悪用または不正使用する意図のあるアプリは一切禁止している。

参考資料

- Google Play デベロッパーポリシーセンター プライバシー、詐欺、デバイスの不正使用

1.5.2.1 ユーザデータ

ユーザデータ（デバイス情報を含む、ユーザについての情報やユーザから収集する情報など）を扱う場合は、その処理方法を明らかにする必要がある。それには、アプリによるユーザデータのアクセス、収集、使用、処理、共有の方法を示すとともに、データの使用をポリシーに準拠した目的に制限することが求められる。ユーザの個人情報と機密情報の扱いについては、さらに下記の「ユーザの個人情報と機密情報」に示す要件も適用されることにご注意すること。これらの Google Play の要件は、プライバシー保護とデータ保護に関する適用法令が規定する要件に加えて適用される。

サードパーティのコード（SDK など）をアプリに含める場合には、アプリ内で使用するサードパーティのコードと、アプリでのサードパーティによるユーザデータの扱いが、使用と開示に関する要件を含め、Google Play デベロッパープログラムポリシーに準拠していることを確認する必要がある。たとえば、SDK プロバイダがアプリを通じてユーザの個人情報や機密情報を販売しないようにする必要がある。この要件は、ユーザデータがサーバに送信された後で転送される場合にも、サードパーティのコードをアプリに埋め込むことで転送される場合にも適用される。

ユーザの個人情報と機密情報

ユーザの個人情報や機密情報には、個人を特定できる情報、財務情報、支払い情報、認証情報、電話帳、連絡先、デバイスの位置情報、SMS や通話に関するデータ、健康に関するデータ、Health Connect のデータ、デバイス上の他のアプリの一覧、マイクやカメラなどのデバイスや使用状況に関するその他の機密情報が含まれますが、これらに限定される。アプリがユーザの個人情報や機密情報を扱う場合は、以下の要件を満たす必要がある。

- アプリを通じて取得した個人情報や機密情報のアクセス、収集、使用、および共有を、ユーザが合理的に予期する目的に適合するアプリとサービスの機能、およびポリシーにのみ許可すること。
  - ユーザの個人情報や機密情報を使用して広告を配信するアプリは、Google Play の広告ポリシーに準拠する必要がある。
  - サービスプロバイダが必要とする場合、あるいは政府機関による有効な要請や適用される法律を遵守するため、もしくは合併または買収の一環として必要な場合には、法的に適切な通知を行ったうえでデータを転送できる。
- 最新の暗号手法を使用して（HTTPS 経由などで）転送するなど、ユーザのすべての個人情報や機密情報を安全に扱うこと。
- Android の権限によって制限されているデータにアクセスする前に、可能な限り実行時の権限をリクエストすること。
- ユーザの個人情報や機密情報を販売しないこと。
  - 「販売」とは、金銭的対価を目的に第三者との間でユーザの個人情報や機密情報の交換または転送を行うことを意味する。
    * ユーザの個人情報や機密情報をユーザが転送すること（たとえば、ユーザがアプリの機能を使用して特定のファイルを第三者に転送したり、調査研究専用のアプリを使用したりすること）は、販売とは見なされない。

認識しやすい開示と同意の要件

アプリによるユーザの個人情報や機密情報のアクセス、収集、使用、共有が、対象のプロダクトや機能のユーザが合理的に予測できる範囲を超えている場合（たとえばユーザがアプリを操作していないときに、データの収集がバックグラウンドで行われるなど）には、以下の要件を満たす必要がある。

認識しやすい開示: データの収集、使用、共有について、アプリ内で開示する必要がある。アプリ内での開示に関する要件は次のとおり。

- アプリ内で開示すること。アプリの説明文やウェブサイトでの開示だけでは不十分である。
- アプリの通常使用時に表示すること。表示するのにメニューや設定に移動する必要のある開示方法では不十分である。
- アクセスまたは収集するデータの種類について説明すること。
- データをどのように使用、共有するかについて説明すること。
- 掲載場所を、プライバシー ポリシーや利用規約のみとしないこと。
- 個人情報や機密情報の収集に関係のない他の開示の中に掲載しないこと。

同意およびランタイム権限: アプリ内でのユーザによる同意のリクエストや、ランタイム権限のリクエストを行う場合は、その直前にこのポリシーの要件に沿ってアプリ内で開示される必要がある。同意を求める場合は、以下のようにする必要がある。

- 同意ダイアログは、あいまいにならないよう明確に表示する。
- 同意を示すための明確な操作をユーザに求める（例: タップで同意する、チェックボックスをオンにする）。
- 開示画面から他へ移動する操作（例: タップで移動する、戻るボタンやホームボタンを押す）を同意と見なさない。
- ユーザの同意を得る方法として、自動で非表示になるメッセージや閲覧期限付きメッセージを使用しない。
- アプリがユーザの個人情報と機密情報の収集やアクセスを開始するには、事前にユーザの許可を得る必要がある。

他の法的根拠（EU の GDPR における正当な利益など）に基づいてユーザの同意なく個人情報と機密情報を処理するアプリは、適用されるすべての法的要件を遵守するとともに、ユーザに対して、このポリシーで要求されるアプリ内開示を含む適切な開示を行う必要がある。

ポリシーの要件に準拠するには、認識しやすい開示に関する以下のサンプル フォーマットを必要に応じて参照することをおすすめする。

- 「[このアプリ] は、[機能] を可能にするために、[想定される状況]、[データの種類] を [収集 / 転送 / 同期 / 保存] する。」
- 例:「Fitness Funds は、フィットネスの記録を可能にするために、アプリが閉じているときや使用されていないときでも、位置情報を収集する。また、位置情報は広告をサポートするためにも使用される。」

- 例:「Call buddy は、組織への連絡を可能にするために、アプリが使用されていないときでも、通話履歴の書き込みと読み込みのデータを収集する。

デフォルトでユーザの個人情報と機密情報を収集するように設計されているサードパーティのコード（SDK など）がアプリに統合されている場合は、Google Play による要請を受けてから 2 週間以内に（Google Play によってそれより長い期間が与えられている場合はその期間内に）、アプリがこのポリシーの認識しやすい開示と同意の要件（サードパーティのコードを通じたデータのアクセス、収集、使用、共有に関する要件を含む）を満たしていることを示す、十分な根拠を提示する必要がある。

違反の例

- デバイスの位置情報を収集するにもかかわらず、このデータを使用する機能と、バックグラウンドでのアプリの使用について、認識しやすい開示で説明していないアプリ。
- データの使用目的を説明する認識しやすい開示が提示される前に、データへのアクセスを要求する実行時の権限が付与されているアプリ。
- ユーザのインストール済みアプリの一覧にアクセスできるにもかかわらず、そのデータを個人情報や機密情報として扱わず、上記のプライバシー ポリシー、データの処理、認識しやすい方法での開示、および同意の各要件を満たしていないアプリ。
- ユーザの電話機能や連絡帳のデータにアクセスできるにもかかわらず、そのデータを個人情報や機密情報として扱わず、プライバシー ポリシー、データの処理、認識しやすい方法での開示、および同意の各要件を満たしていないアプリ。
- ユーザの画面を記録するにもかかわらず、そのデータを個人情報や機密情報として、このポリシーに沿って扱わないアプリ。
- デバイスの位置情報を収集するにもかかわらず、上記の要件に沿ってその情報の使用について包括的に開示せず、同意を得ていないアプリ。
- 追跡、調査、またはマーケティングの目的などのため、アプリのバックグラウンドで制限付きの権限を使用するにもかかわらず、上記の要件に沿って各権限の使用について包括的に開示せず、同意を得ていないアプリ。
- ユーザの個人情報と機密情報を収集し、そのデータをこのユーザデータ ポリシーや、アクセス、データ処理（許可されていない販売を含む）、認識しやすい開示と同意の要件に沿って処理しない SDK を使用するアプリ。

認識しやすい開示と同意の要件について詳しくは、この記事を確認すること。

個人情報と機密情報へのアクセスに関する制限

上記の要件に加えて、特定の操作における要件を下表に記載する。

表 1.5.2.1.1　ユーザデータの特定の操作における要件

操作	要件
個人の財務情報、支払い情報、政府発行の個人識別番号をアプリが扱う場合	財務処理、支払い処理、政府発行の個人識別番号に関する個人情報や識別情報は一切公開 してはならない。
非公開の電話帳や連絡先情報をアプリが扱う場合	個人の非公開の連絡先を許可なく公開または開示することは認められない。
ウイルス対策、マルウェア対策、セキュリティ関連の機能など、ウイルス対策機能やセキ ュリティ機能を持つアプリの場合	アプリ内での開示と併せて、アプリが収集、転送するユーザデータの種類と内容、使用 方法、共有先について説明するプライバシー ポリシーを掲載する必要がある。
アプリが子供を対象とする場合	子供向けサービスで使用が承認されていない SDK をアプリに含めてはならない。ポリシー のすべての文言と要件については、子供向けやファミリー向けにアプリを設計するを確認すること。
永続的なデバイス識別子（IMEI、IMSI、SIM のシリアル番号など）を収集またはリンクするアプリの場合	永続的なデバイス識別子を、他の個人情報と機密情報、またはリセット可能なデバイス識別子にリンクしてはならない。ただし、以下を目的とする場合を除く。 ・SIM 識別子にリンクされた通話機能（例: 携帯通信会社アカウントにリンクされた Wi-Fi 通話機能） ・デバイス所有者モードを使用するエンタープライズデバイス管理アプリ これらの使用方法は、ユーザデータに関するポリシーの規定に沿って、ユーザが認識しやすいように開示する必要がある。 その他の一意の識別子については、こちらのリソースを確認すること。 Android 広告 ID に関する追加のガイドラインについては、広告ポリシーを参照すること。

データ セーフティ セクションすべてのデベロッパーは、すべてのアプリについて、ユーザデータの収集、使用、共有に関する詳細な説明を、データセーフティセクションに明瞭かつ正確に記載する必要がある。デベロッパーには、ラベルを正確に記載し、ラベルの情報を最新の状態に保つ責任がある。データセーフティセクションは、該当箇所において、アプリのプライバシーポリシーで開示されている内容と一致する必要がある。

データセーフティセクションへの入力に関する追加情報については、こちらの記事を参照すること。

プライバシー ポリシー

すべてのアプリで、プライバシーポリシーのリンクを Google Play Console 内の所定の欄に掲載し、アプリ内にはプライバシーポリシーのリンクまたはテキストを掲載する必要がある。プライバシーポリシーでは、アプ

リ内での開示内容と併せて、当該アプリでユーザデータ（プライバシー ラベルで開示されているデータに限定されない）がどのようにアクセス、収集、使用、共有されるかを包括的に開示する必要がある。これには以下の情報が含まれる。

- デベロッパー情報、およびプライバシーに関する連絡先または問い合わせを行う方法。
- アプリがアクセス、収集、使用、共有するユーザの個人情報や機密情報の種類、およびユーザの個人情報や機密情報の共有先の開示。
- ユーザの個人情報や機密情報を安全に処理するための手順。
- デベロッパーのデータ保持ポリシーおよびデータ削除ポリシー。
- プライバシー ポリシーであることが明瞭にわかるラベル付け（たとえば、タイトルに「プライバシー ポリシー」と記載する）。

アプリの Google Play ストアの掲載情報に記載されている主体（デベロッパーや会社等）がプライバシーポリシーに明記されている、もしくはアプリ名がプライバシー ポリシーに明記されている必要がある。ユーザの個人情報や機密情報にアクセスしないアプリであっても、プライバシー ポリシーを掲載する必要がある。

プライバシーポリシーは必ず、どの国からもアクセスできるよう、アクセス制限のない一般公開の有効な URL（PDF ではない）で参照可能、かつ編集不可にすること。

アプリセット ID の使用

Android では、分析や不正防止などの重要なユースケースに対応するために、新しい ID が導入される。この ID を使用するための規約は以下のとおり。

- 使用: アプリセット ID を広告のパーソナライズと広告の測定に使用することはできない。
- 個人を特定できる情報またはその他の識別子との関連付け: 広告を目的として、アプリセット ID を Android の識別子（例: AAID）または個人情報や機密情報に関連付けることはできない。
- 透明性と同意: アプリセット ID を収集および使用することと、この規約を遵守していることを、法的に適切なプライバシーに関するお知らせ（デベロッパー独自のプライバシー ポリシーを含む）を通じてユーザに開示する必要がある。必要に応じて、ユーザから法的に有効な同意を得る必要があります。Google のプライバシー基準について詳しくは、ユーザデータに関するポリシーを確認すること。

EU-U.S., Swiss Privacy Shield（EU-US スイス プライバシー シールド）

Google が公開している、欧州連合またはスイスにおいて収集された直接または間接的に個人を特定できる個人情報（「EU 個人情報」）にアクセスする場合や、そうした個人情報を利用、処理する場合は、以下の義務がある。

- 適用のある法域におけるプライバシー、データ セキュリティ、データ保護に関するあらゆる法律、指令、規制、ルールを遵守すること
- EU 個人情報のアクセス、使用、処理は、その EU 個人情報に関連する人物が同意した目的の範囲内に限って行うこと
- データの消失、不正使用、不正または違法アクセス、漏えい、改変、破壊などから EU 個人情報を保護するために適切な組織的および技術的な措置をとること

- Privacy Shield（プライバシー シールド）原則で要求されているものと同水準の保護を確保すること

上記の義務を遵守していることを定期的に監視し、上記の条件を満たせない（または満たせなくなるリスクが高い）場合は、直ちに data-protection-office@google.com 宛てのメールで Google に通知するとともに、直ちに EU 情報の処理を停止するか、適切な水準の保護を確保するための合理的かつ適切な措置を講じなければならない。

2020 年 7 月 16 日をもって、Google では、欧州経済領域または英国から米国へのデータ転送において EU-U.S. Privacy Shield（EU-US プライバシー シールド）の利用を終了した（詳細）。詳しくは、デベロッパー販売 / 配布契約の第 9 条を参照。

参考資料

- Google Play デベロッパーポリシーセンター ユーザデータ

1.5.2.2 機密情報へのアクセスに関する権限と API

機密情報にアクセスする権限や API のリクエストは、ユーザにとって理に適うものでなければならない。そのため、機密情報にアクセスする権限や API をリクエストできるのは、アプリで現在提供している機能やサービスの実装に必要で、それらが Google Play ストアの掲載情報に掲載されている場合に限られる。ユーザデータやデバイスデータへのアクセスを必要とする機能や目的が公開されていないもしくは実装されていない場合、または認可されていない場合には、機密情報にアクセスする権限や API は利用できない。機密情報にアクセスする権限または API を通じてアクセスした個人情報や機密情報を販売したり、販売を促進する目的で共有したりすることは禁止されている。

データにアクセスするために、機密情報にアクセスする権限または API をリクエストする場合は、アプリが権限をリクエストする理由をユーザが理解しやすいように状況に合わせて（段階的にリクエストする形で）行うようにする。データの使用は、ユーザが同意した目的に限って行わなければならない。後になって他の目的でデータを使用する必要が出てきた場合は、その追加の用途に関して、あらためてユーザにリクエストし、同意を得る必要がある。

制限付きの権限

上記に加えて、制限付きの権限とは、「Dangerous」、「Special」、「Signature」と指定される権限、または下記のような権限である。これらの権限には、以下に示す追加の要件と制限が適用される。

- 制限付きの権限を通じてアクセスされたユーザデータやデバイスデータは、ユーザの個人情報および機密情報と見なされ、ユーザデータに関するポリシーの要件が適用される。
- ユーザが制限付き権限のリクエストを承認しない場合はその決定を尊重すること。重要でない権限についてユーザに同意するよう誘導または強制することも認められない。機密情報に関わる権限へのアクセスを許可しないユーザにも対応する（たとえば、ユーザが通話履歴へのアクセスを制限している場合であれば電話番号を手動で入力できるようにするなど）よう、合理的な努力を尽くす必要がある。
- Google Play のマルウェアに関するポリシー（昇格させた権限の悪用を含む）に違反する権限の使用は、明示的に禁止されている。

一部の制限付き権限には、下記の追加要件が適用されることがある。これらの制限の目的は、ユーザのプライバシーを守ることにある。ただし、非常にまれなケースですが、アプリがきわめて必要性の高いまたは重要な

機能を提供していて、その機能を実現する他の手段が現時点で他に存在しない場合には、下記の要件について例外が認められることがある。例外の申請があった場合は、ユーザに対して想定されるプライバシーまたはセキュリティ上の影響を考慮して審査する。

SMS と通話履歴の権限

SMS と通話履歴に関する権限は、ユーザの個人情報と機密情報と見なされ、個人情報と機密情報に関するポリシーおよび以下の制限が適用される。

表 1.5.2.2.1　制限付きの権限と要件

制限付きの権限	要件
通話履歴に関する権限グループ（例: READ_CALL_LOG、WRITE_CALL_LOG、PROCESS_OUTGOING_CALLS）	ユーザのデバイスで、アプリがデフォルトの電話ハンドラまたはアシスタント ハンドラとして能動的に登録されている必要がある。
SMS に関する権限グループ（例: READ_SMS、SEND_SMS、WRITE_SMS、RECEIVE_SMS、RECEIVE_WAP_PUSH、RECEIVE_MMS）	ユーザのデバイスで、アプリがデフォルトの SMS ハンドラまたはアシスタント ハンドラとして能動的に登録されている必要がある。

デフォルトの SMS ハンドラ、電話ハンドラまたはアシスタントハンドラとしての機能をアプリが備えていない場合、マニフェストで上記の権限の使用を宣言することはできない（マニフェスト内のプレースホルダ テキストである場合を含む）。また、ユーザに上記の権限の許可をリクエストする前に、アプリがデフォルトの SMS ハンドラ、電話ハンドラまたはアシスタントハンドラとして有効に登録されている必要がある。アプリがデフォルトのハンドラではなくなったときは、直ちに該当する権限の使用を停止しなければならない。許可されている使用方法と例外については、ヘルプセンターのこちらのページを確認すること。

アプリが使用できる権限は、承認済みの重要なアプリの機能を提供するために必要な権限（およびその権限で取得したデータ）のみである。重要な機能とは、アプリの主たる目的である機能を言う。いくつかの機能のセットである場合もあり、そのすべてがアプリの説明文の中に認識しやすい形で明記されている必要がある。その機能がなければアプリが「壊れている」、使用できない、と見なされるような機能が「重要な機能」にあたる。このデータの転送、共有、またはライセンス下での使用は、アプリ内で重要な機能やサービスを提供することのみを目的として許可されるものであり、その他の目的（他のアプリやサービスの改善、広告、マーケティング目的など）でデータを使用することはできない。通話履歴や SMS に関連する権限に基づくデータを取得するために、他の権限、API、第三者の提供元など、代わりの方法を使用することはできない。

位置情報の利用許可

デバイスの位置情報は、ユーザの個人情報および機密情報と見なされ、個人情報と機密情報に関するポリシー、バックグラウンドでの位置情報に関するポリシー、および以下の要件が適用される。

- アプリで現在の機能やサービスを提供する必要がなくなった後は、位置情報の権限（ACCESS_FINE_LOCATION、ACCESS_COARSE_LOCATION、ACCESS_BACKGROUND_LOCATION など）で保護されているデータにアプリからアクセスすることはできない。
- 広告や分析のみを目的として、ユーザに位置情報の利用許可をリクエストしてはなりません。このデータの許可された利用の範囲を広告配信にも適用するアプリは、広告ポリシーを遵守していなければならない。

- アプリがリクエストするアクセスのレベルは、位置情報を必要とする現在の機能やサービスを提供するうえで必要最低限に留め（つまり、高精度よりも低精度、バックグラウンドよりもフォアグラウンド）、そのレベルの位置情報が機能やサービスに必要であることをユーザが合理的に予期できる必要がある。たとえば、バックグラウンドで位置情報をリクエストまたは利用することに合理的な根拠がないアプリは承認されない可能性がある。
- バックグラウンドで位置情報が利用できるのは、ユーザにとってメリットがあり、かつアプリの重要な機能に関連する機能を提供する場合のみである。

アプリからフォアグラウンド サービスの権限で（たとえば「使用中」のみ許可されるなど、フォアグラウンドでのアクセス権でのみ）位置情報にアクセスするのが認められるのは、以下の場合である。

- アプリ内でユーザが開始したアクションの続きとして位置情報の利用が開始され、かつ
- ユーザが開始したアクションの意図されたユースケースが完了した後、直ちにその利用が終了する場合

子供を主な対象とするアプリは、ファミリー向けポリシーを遵守する必要がある。

ポリシーの要件について詳しくは、こちらのヘルプ記事を確認する。

すべてのファイルへのアクセス権限

ユーザのデバイス上のファイルとディレクトリの属性は、ユーザの個人情報および機密情報と見なされ、個人情報と機密情報に関するポリシーおよび以下の要件が適用される。

- アプリがリクエストするアクセスの対象は、アプリが機能するうえで不可欠なデバイス ストレージに限定する必要がある。ユーザ向けの不可欠なアプリ機能と関係のない目的で第三者のためにデバイス ストレージへのアクセスをリクエストしてはならない。
- R 以降を搭載している Android デバイスでは、共有ストレージ内のアクセスを管理するには、MANAGE_EXTERNL_STORAGE 権限が必要である。R をターゲットとし、共有ストレージへの幅広いアクセス（「すべてのファイルへのアクセス」）をリクエストするアプリは必ず、公開前に適切なアクセスに関する審査に合格する必要がある。この権限の使用を許可されたアプリは、[特別なアプリ アクセス] 設定で [すべてのファイルへのアクセス] を有効にするように求めるメッセージを、ユーザにはっきり表示する必要がある。R のこの要件について詳しくは、こちらのヘルプ記事を確認する。

パッケージ（アプリ）の公開設定権限

デバイスにクエリして入手したインストール済みアプリのインベントリは、ユーザの個人情報および機密情報と見なされ、個人情報と機密情報に関するポリシーおよび以下の要件が適用される。

デバイス上の他のアプリを起動、検索、相互運用することが主要な目的であるアプリは、以下で概説するように、デバイス上の他のインストール済みアプリに対して、スコープに応じた可視性を得ることができる。

- **広範なアプリの可視性**: 広範な可視性とは、アプリがデバイス上のインストール済みアプリ（「パッケージ」）を幅広く（「広範に」）見渡せる（可視性が与えられている）ことを指す。
  - API レベル 30 以降をターゲットとするアプリの場合、QUERY_ALL_PACKAGES 権限によってインストール済みアプリについて広範な可視性が得られるのは、特定のユースケース（当該アプリが機能するためにデバイス上のすべてのアプリを認識するか、それらのアプリと相互運用する必要がある）に制限される。

    * スコープを絞ったパッケージの可視性を宣言（広範な可視性をリクエストせず、特定のパッケージをクエリしてやり取りするなど）して動作させることが可能なアプリでは、QUERY_ALL_PACKAGES を使用しない。

  – QUERY_ALL_PACKAGES 権限に関連する広範な可視性レベルに近い別の方法の使用も同様に、ユーザ向けの主要なアプリ機能と、その別の方法によって検出されたアプリとの相互運用に制限される。

  – QUERY_ALL_PACKAGES 権限を使用できるユースケースについては、こちらのヘルプセンター記事を確認する。

- **限定的なアプリ公開設定**: 限定的な公開設定とは、アプリが、よりターゲットを絞った（「広範」ではない）方法を使って特定のアプリのクエリを行うことによりデータへのアクセスを最小限に抑える場合（アプリのマニフェスト宣言を満たす特定のアプリのクエリを行う場合など）を指す。アプリが相互運用性に関するポリシーを遵守している場合や他のアプリの管理を担っている場合は、この方法でそれらのアプリのクエリを行える。

- デバイス上のインストール済みアプリのインベントリに対する公開設定は、アプリの主要な目的やユーザがアプリ内でアクセスする主要な機能に直接関連する必要がある。

Play で配信中のアプリにクエリして入手したアプリインベントリのデータを、分析や広告収益化の目的で販売、共有することは禁止されている。

Accessibility API

Accessibility API を次の目的で使用することはできない。

- ユーザの許可なくユーザ設定を変更したり、ユーザがアプリまたはサービスを無効化またはアンインストールできないようにしたりする。ただし、保護者による使用制限を使用するアプリを通じて親権者または保護者の許可を得るか、エンタープライズ マネジメント ソフトウェアを通じて認定済み管理者の許可を得た場合を除く。

- Android の組み込みのプライバシー管理とプライバシー通知を回避する。

- 不正な方法または Google Play デベロッパー ポリシーに違反するその他の方法で、ユーザ インターフェースを変更または利用する。

Accessibility API は、リモート通話の音声録音用には設計されておらず、そのようなリクエストを受けることもできない。

Accessibility API を使用する場合は、Google Play のストアの掲載情報に記載する必要がある。

IsAccessibilityTool に関するガイドライン

障がいのあるユーザを直接サポートする機能が主体になっているアプリは、IsAccessibilityTool を使用して、ユーザ補助アプリとして公式に表明できる。

IsAccessibilityTool の対象とならないアプリはこのフラグを使用できないが、ユーザ補助に関連する機能をユーザが認識しにくいため、ユーザデータに関するポリシーに規定されている「認識しやすい開示と同意」の要件を遵守する必要がある。詳しくは、AccessibilityService API のヘルプセンター記事を確認する

Accessibility API を使用しなくても必要な機能を提供できるのであれば、より限定された範囲の API と権限をアプリで使用すること。

パッケージ インストールのリクエスト（REQUEST_INSTALL_PACKAGES）権限

REQUEST_INSTALL_PACKAGES 権限を使用すると、アプリからアプリ パッケージのインストールをリクエストできる。この権限を使用するには、アプリのコア機能に以下が含まれている必要がある。

- アプリ パッケージを送信または受信する機能、および
- ユーザが自発的にアプリ パッケージのインストールを開始する機能

たとえば次のような機能が許可されています。

- ウェブのブラウジングまたは検索
- 添付ファイルをサポートするコミュニケーションサービス
- ファイルの共有、転送、管理
- 企業向けデバイスの管理
- バックアップと復元
- デバイスの移行または電話の転送

コア機能とは、アプリの主要な目的を果たすために必要不可欠な機能を指し、そのすべてがアプリの説明文に認識しやすい形で明記されている必要がある。

REQUEST_INSTALL_PACKAGES 権限は、デバイス管理を目的とする場合を除き、自動更新、修正、アセット ファイル内での他の APK のビルドには使用できない。パッケージの更新とインストールにおいては、常に Google Play のデバイスやネットワークでの不正行為に関するポリシーに準拠しなければならず、ユーザが自発的に操作する必要がある。

Android の権限によるヘルスコネクト

ヘルスコネクト権限を通じてアクセスされたデータは、ユーザの個人情報および機密情報と見なされ、ユーザデータに関するポリシーおよび以下の追加要件が適用される。

ヘルスコネクトのアクセス方法と使用方法

ヘルスコネクトを通じてデータにアクセスするためのリクエストは、明確にわかりやすく記述すること。ヘルスコネクトは、該当するポリシーと利用規約を遵守したうえで、このポリシーによって規定されている承認された用途に限り使用できる。これはつまり、権限へのアクセスをリクエストできるのは、アプリまたはサービスの用途が、承認されているいずれかの用途に該当する場合に限られることを意味する。

ヘルスコネクト権限へのアクセスを承認される用途は次のとおり。

- ユーザの健康とフィットネスにとって有益な 1 つ以上の機能をユーザ インターフェースを通じて利用できるアプリまたはサービス。ユーザが自分の身体活動、睡眠、心身の健康、栄養、健康状態の測定値、身体的特徴、および / または健康とフィットネスに関連するその他の記述や測定値を直接記録、レポート、モニタリング、および / または分析できる。

- ユーザの健康とフィットネスにとって有益な 1 つ以上の機能をユーザ インターフェースを通じて利用できるアプリまたはサービス。ユーザが自分の身体活動、睡眠、心身の健康、栄養、健康状態の測定値、身体的特徴、および / または健康とフィットネスに関連するその他の記述や測定値をスマートフォンおよび / またはウェアラブルに保存して、用途に適合するその他のオンデバイス アプリとデータを共有できる。

ヘルスコネクトは、ユーザが Android デバイス内のさまざまなソースから健康とフィットネスに関するデータを集めて、特定の第三者と共有できる、汎用のデータ ストレージおよびデータ共有プラットフォームである。データは、ユーザが任意に選択したソースから集めることができる。デベロッパーは、ヘルスコネクトが意図する用途に適しているかを評価するとともに、特定の目的に関連して、また特に調査、健康、または医療上の用途について、ヘルスコネクトからのデータのソースと質を精査する必要がある。

- ヘルスコネクトを通じて取得したデータを使用して、健康関連の被験者調査を実施するアプリは、被験者（未成年者の場合は保護者）の同意を得る必要がある。そのような同意事項には、(a) 調査の性質、目的、期間、(b) 調査手順、被験者に対するリスクおよび利点、(c) データの機密性および取り扱い（第三者との共有を含む）に関する情報、(d) 被験者の質問に対応する問い合わせ先、(e) 取り消し手順が含まれるものとする。ヘルスコネクトを通じて取得したデータを使用して、健康関連の被験者調査を実施するアプリは、(1) 被験者の権利、安全、心身の健康を保護する目的を持ち、(2) 被験者調査を精査、変更、承認する権限を有する、独立した委員会による承認を得る必要がある。要請があった場合には、そのような承認の証明を提出しなければならない。

- デベロッパーは、ヘルスコネクトの用途、およびヘルスコネクトを通じて得られたデータの用途に基づいて適用される、規制または法律上の要件を遵守する責任がある。Google の特定のプロダクトまたはサービスについて、Google が提供するラベルまたは情報に明示的に記載されていない限り、Google は、特に調査、健康、医療用途に限らず、いかなる用途または目的でも、ヘルスコネクトに含まれるデータの使用を推奨し、その正確性を保証することはありません。Google は、ヘルスコネクトを通じて取得されたデータの使用に関連する、いかなる責任も負わない。

限定的な使用

適切な用途でヘルスコネクトを使用する際には、ヘルスコネクトを通じてアクセスするデータも、以下の要件を遵守して使用する必要がある。これらの要件は、ヘルスコネクトから取得した元データと、元データから集計、匿名化、または取得されたデータに適用される。

- ヘルスコネクトのデータの使用は、リクエスト元アプリのユーザ インターフェースに明確に表示される、適切な用途または機能の提供もしくは改善に限定される。

- 以下の目的がある場合を除き、ユーザデータを第三者に譲渡してはならない。

  - ユーザの同意に基づき、リクエスト元アプリのユーザ インターフェースに明確に表示される、適切な用途または機能を提供もしくは改善する場合。

  - セキュリティ上の目的で必要な場合（不正使用の調査など）。

  - 適用される法律および / または規制を遵守するために必要な場合。

  - ユーザから事前に明示的な同意を得た後で、デベロッパーの合併、買収、または資産売買の一環として行う場合。

- 以下の場合を除き、人にユーザデータが読まれないようにする必要がある。
  - 特定のデータを読まれることに、ユーザが明示的に同意している場合。
  - セキュリティ上の目的で必要な場合（不正使用の調査など）。
  - 適用される法律を遵守するために必要な場合。
  - データ（派生データを含む）を集計し、適用されるプライバシー要件および地域のその他の法的要件を遵守した、内部オペレーションのために使用する場合。

ヘルスコネクト データのその他の譲渡、使用、または販売は、以下の行為を含めてすべて禁止されている。

- 広告プラットフォーム、データ ブローカー、または情報リセラーなどの第三者にユーザデータを譲渡または販売すること。
- パーソナライズ広告やインタレスト ベース広告など、広告の配信を目的としてユーザデータを譲渡、販売、または使用すること。
- 信用力を判断するため、または貸与目的でユーザデータを譲渡、販売、または使用すること。
- 連邦食品・医薬品・化粧品法のセクション 201(h) に基づく医療機器と見なされるプロダクトまたはサービスを使用して、規制対象の機能を実行するために、ユーザデータを譲渡、販売、または使用すること。
- Google から書面による事前の承認を得ている場合を除き、（HIPAA によって定義される）保護対象保健情報に関連する目的で、方法を問わず、ユーザデータを譲渡、販売、または使用すること。

このポリシー、またはヘルスコネクトについて適用されるその他の利用規約またはポリシーに違反する形で、ヘルスコネクトにアクセスしてはならない。これには以下を目的とする場合が含まれる。

- ヘルスコネクトの使用または障害によって、死亡、人身傷害、もしくは環境上または財産上の損害に至ることが合理的に予想されるようなアプリ、環境、またはアクティビティ（核施設、航空管制システム、生命維持装置、兵器の作成または操作など）については、その開発、あるいはそれらに組み込む目的で、ヘルスコネクトを使用してはならない。
- ヘルスコネクトを通じて取得したデータに、ヘッドレス アプリを使用してアクセスしてはならない。アプリでは、アプリトレイ、デバイスのアプリ設定、通知アイコンなどに、明確に特定できるアイコンを表示する必要がある。
- 対応していないデバイスまたはプラットフォーム間でデータを同期するアプリで、ヘルスコネクトを使用してはならない。
- 子供だけを対象としているアプリ、サービス、または機能にヘルスコネクトを接続することはできない。主に子供を対象としているサービスでヘルスコネクトを使用することは承認されていない

ヘルスコネクトのデータの使用が、限定的な使用に関する制限を遵守していることを示す確認的陳述書を、アプリ内、あるいはウェブサービスまたはアプリに関連するウェブサイト上で開示する必要がある。たとえばホームページで、専用ページまたはプライバシー ポリシーに関する注記へのリンクを示し、「ヘルスコネクトから受け取った情報の使用については、限定的な使用に関する要件を含む、ヘルスコネクト権限ポリシーを遵守してください」などと記載する。

最小範囲

アクセス権限をリクエストできるのは、アプリまたはサービスの機能を実装するために不可欠な場合に限る。

これは次のことを意味する。

- 必要のない情報へのアクセス権限はリクエストしないこと。プロダクトの機能またはサービスの実装に必要なアクセス権限のみリクエストできる。プロダクトで特定のアクセス権限を必要としない場合、そのアクセス権限はリクエストしないこと。

通知と管理の透明性と正確性

ヘルスコネクトは健康とフィットネスに関するデータを扱うが、それには個人情報と機密情報が含まれる。すべてのアプリとサービスについてプライバシー ポリシーを規定し、アプリまたはサービスがユーザデータを収集、使用、共有する方法を包括的に開示する必要がある。開示する情報には、ユーザデータを共有する当事者の種類、データの使用方法、データの保存と保護の方法、アカウントが無効になるか削除された場合のデータの扱いが含まれるものとする。

適用される法律で規定されている要件に加えて、デベロッパーは以下の要件を遵守する必要がある。

- データのアクセス、収集、使用、共有について開示する必要があります。開示については次のことが求められる。
  - ユーザデータへのアクセスを求めるアプリまたはサービスの識別情報を正確に示す。
  - アクセス、リクエスト、および / または収集するデータの種類に関して、明確かつ正確な情報を提供する。
  - データを使用、共有する方法を示す。1 つの目的でデータをリクエストしながら、別の目的でもデータを使用する場合には、ユーザに両方の用途を通知する必要がある。
- ユーザがアプリ上で個人データを管理または削除する方法を示すヘルプ ドキュメントを提供する。

安全なデータ処理

ユーザデータはすべて安全に扱う必要があります。デベロッパーは合理的かつ適切な手順に沿って、ヘルスコネクトを使用するすべてのアプリケーションまたはシステムを、不正または違法なアクセス、使用、破壊、紛失、改変、開示から保護する必要がある。

推奨されるセキュリティ対策としては、たとえば ISO/IEC 27001 などで規定されている情報セキュリティ管理システムを実装して維持することで、アプリまたはウェブサービスを堅牢にし、OWASP トップ 10 に示されているセキュリティ上の一般的な問題がない状態を確保することが挙げられる。

デベロッパーのプロダクトによってユーザが所有するデバイスからデータが転送される場合には、使用している API、ユーザによる権限付与の数、またはユーザ数に応じて、アプリまたはサービスについて定期的なセキュリティ評価を受け、指定した第三者による評価文書を取得する必要がある。

ヘルスコネクトに接続するアプリに関する要件について詳しくは、こちらのヘルプ記事を確認する。

VPN サービス

VpnService は、アプリが独自の VPN ソリューションを拡張、構築できるようにするための基本クラスである。VPN がコア機能であり、VpnService を使用するアプリのみが、リモート サーバへのデバイスレベルのセキュ

アなトンネルを作成できる。ただし、次のようなコア機能を実装するためリモート サーバを必要とするアプリは例外となる。

- 保護者による使用制限や企業による管理を実装するアプリ。
- アプリの使用状況をトラッキングするアプリ。
- デバイス保護アプリ（ウイルス対策、モバイル デバイス管理、ファイアウォールなど）。
- ネットワーク関連ツール（リモート アクセスなど）。
- ウェブ ブラウジング用アプリ。
- テレフォニーサービスまたは接続サービスを提供するために VPN 機能の使用を必要とする携帯通信会社のアプリ。

VpnService を次の目的で使用することはできない。

- 認識しやすい開示および同意機能を実装せずに、ユーザの個人情報や機密情報を収集する。
- 収益化を目的として、デバイスでの他のアプリからのユーザ トラフィックをリダイレクトまたは操作する（ユーザの国とは異なる国から広告トラフィックをリダイレクトするなど）。
- アプリの収益化に影響を与えられるように広告を操作する。

VpnService を使用するアプリは、次のすべての要件を満たす必要がある。

- VpnService を使用することを Google Play の掲載情報に記載する。
- デバイスから VPN トンネル エンドポイントに送信されるデータを暗号化する。
- 広告の不正行為、権限、マルウェアに関するポリシーを含む、すべてのデベロッパープログラムポリシーに準拠する。

参考資料

- Google Play デベロッパーポリシーセンター 機密情報へのアクセスに関する権限と API

1.5.2.3 デバイスやネットワークでの不正行為

ユーザのデバイスやその他のデバイス、パソコン、サーバ、ネットワーク、アプリケーション プログラミング インターフェース（API）、サービスなど（デバイス上の他のアプリ、Google サービス、許可された携帯通信会社のネットワークを含む）を妨害、阻害、破損する、またはそれらに無断でアクセスするアプリは認められない。

Google Play に公開するアプリは、Google Play のアプリの中核品質に関するガイドラインに定めるデフォルトの Android システム最適化要件を遵守している必要がある。

Google Play で販売または配布されるアプリについては、Google Play の更新機能以外の方法によりアプリ自体の変更、差し替え、更新を行うことはできない。同様に、Google Play 以外の提供元から実行コード（dex、JAR、.so などのファイル）をダウンロードすることもできない。この制限は、Android API への間接アクセス

を提供する仮想マシンまたはインタープリタ（WebView またはブラウザ内の JavaScript など）で実行されるコードには適用されない。

アプリまたはサードパーティのコード（SDK など）でインタープリタ言語（JavaScript、Python、Lua など）が実行時に読み込まれる場合（たとえば、アプリにパッケージされていない場合）、それらが Google Play ポリシーに違反する可能性があってはならない。

セキュリティの脆弱性を組み込むまたは悪用するコードは許可されない。デベロッパーに報告された最近のセキュリティに関する問題については、アプリセキュリティ向上プログラムを確認すること。

FLAG_SECURE の要件

FLAG_SECURE は、アプリのコードで宣言される表示関連のフラグである。アプリの使用中、UI に含まれるセンシティブ データの表示場所をセキュアなサーフェスに限定することを示せる。このフラグは、センシティブ データがスクリーンショットに表示されたり、セキュアでないディスプレイで閲覧されたりするのを防ぐために設計されている。デベロッパーは、アプリのコンテンツがアプリやユーザのデバイスの外部で閲覧されたり、外部にブロードキャストまたは送信されたりできないようにする場合に、このフラグを宣言する。

セキュリティおよびプライバシー保護のため、Google Play で配信されるすべてのアプリが、他のアプリの FLAG_SECURE 宣言を尊重しなければならない。つまり、他のアプリでの FLAG_SECURE の設定を回避する手段を作成または促進してはいけない。

ユーザ補助ツールとして認められるアプリは、FLAG_SECURE で保護されたコンテンツをユーザのデバイスの外部でアクセスするために転送、保存、またはキャッシュに保存しない限り、この要件の対象外となる。

よくある違反の例

- 広告を表示して他のアプリをブロックまたは妨害するアプリ。
- 他のアプリのゲームプレイに影響を与えるようなゲームチートアプリ。
- サービス、ソフトウェア、ハードウェアのハッキング方法や、セキュリティ保護の回避方法を推進または説明するアプリ。
- サービスや API に対してその利用規約に違反する方法でアクセスまたは利用するアプリ。
- システムの電源管理を迂回しようとするアプリのうち許可リストへの登録が認められないもの。
- 第三者に対するプロキシ サービスの利用を支援するアプリのうち、その利用支援がアプリのユーザ向けの主たる基本目的ではないもの。
- アプリまたはサードパーティのコード（SDK など）のうち、Google Play 以外の提供元から実行可能コード（dex ファイル、ネイティブ コードなど）をダウンロードするもの。
- ユーザの事前の同意なしに他のアプリをデバイスにインストールするアプリ。
- 不正なソフトウェアにリンクするアプリや、その配信やインストールを推進するアプリ。
- アプリまたはサードパーティのコード（SDK など）のうち、信頼できないウェブ コンテンツ（http:// URL）、または信頼できないソースから取得された未検証の URL（信頼できないインテントで取得された URL など）を読み込む JavaScript インターフェースが追加された WebView を含むもの。

参考資料

- Google Play デベロッパーポリシーセンター デバイスやネットワークでの不正行為

1.5.2.4 虚偽の振る舞い

ユーザを欺こうとするアプリや不正行為を助長するアプリは認められない。たとえば、機能的に不可能だと判断されるアプリが含まれますが、これに限定されない。アプリは、メタデータのあらゆる部分で、アプリの機能に関する説明、画像または動画を正確に開示しなければならない。オペレーティング システムや他のアプリの機能または警告であるかのように装うことも認められない。デバイスの設定を変更する場合は、ユーザに通知して同意を得ること、およびユーザが簡単に元に戻せることが必要である。

誤解を与える表現

アプリの説明、タイトル、アイコン、スクリーンショットなどに、虚偽のまたは誤解を招くような情報や宣伝文句を含めたアプリは認められていない。

違反の例

- 機能の説明が誤っている、または不正確でわかりにくいアプリ:
  - アプリの説明やスクリーンショットにはレーシング ゲームのように記載されているのに、実際は自動車の絵のブロックパズル ゲーム。
  - ウイルス対策アプリのように記載されているのに、ウイルスを削除する方法を説明したテキストガイドしか含まれていないアプリ。
- 虫よけアプリなど、実現できない機能を宣伝するアプリ（いたずら、フェイク、冗談として表示されているものも含まれる）。
- レーティングやカテゴリなど（これらに限らず）について、不適切に分類されたアプリ。
- 選挙の投票プロセスに影響を与える可能性のある、明らかに虚偽のコンテンツ。
- 政府機関との協力関係があるように偽るアプリ、あるいは適切な認可を受けずに行政サービスを提供または支援するアプリ。
- 定評のある組織の正式なアプリであるかのように偽るアプリ。「ジャスティン ビーバー公式」のようなタイトルは、必要な許可や権利を得ていない限り、認められまない。

デバイス設定の不正な変更

ユーザの理解や同意を得ずに、アプリ外でユーザのデバイスの設定や機能を変更するアプリは認められない。デバイスの設定や機能には、システムやブラウザの設定、ブックマーク、ショートカット、アイコン、ウィジェット、ホーム画面でのアプリの表示などがある。

その他、次のようなアプリは認められない。

- ユーザの同意を得るが、簡単には元に戻せない方法でデバイスの設定や機能を変更するアプリ。
- サードパーティへのサービスとして、または広告表示を目的として、デバイスの設定や機能を変更するアプリや広告。

- ユーザを欺いて、サードパーティ アプリの削除や無効化、またはデバイスの設定や機能の変更を誘導するアプリ。
- セキュリティ サービスの一環として確認可能な場合を除き、サードパーティ アプリの削除や無効化、またはデバイスの設定や機能の変更をユーザに促したり、報奨付きで奨励したりするアプリ。

不正行為を助長する

人を欺くことを可能にするアプリや機能的に虚偽の振る舞いをするアプリは認められない。たとえば、ID カード、社会保障番号、パスポート、卒業証書、クレジット カード、銀行口座、運転免許証などを偽造できる、または偽造を補助するアプリが該当する（ただし、これらに限定されません）。アプリは、アプリの機能や内容に関してタイトル、説明、画像または動画を正確に開示し、ユーザが期待するとおり合理的に正確に機能しなければならない。

追加のアプリリソース（ゲームアセットなど）は、ユーザによるアプリの利用に不可欠な場合にのみダウンロード可能である。ダウンロードされるリソースは Google Play のすべてのポリシーを遵守する必要がある。また、ダウンロード開始前に、ユーザにメッセージを表示し、ダウンロード サイズを明確に開示する必要がある。

アプリが「いたずら」や「娯楽目的」などである、と主張する場合でも、アプリがポリシーの適用対象外となることはない。

違反の例

- 他のアプリやウェブサイトを装ってユーザに個人情報や認証情報を開示するようにだますアプリ。
- 同意を得ていない個人や団体の、未確認のまたは実在する電話番号、連絡先、住所、個人情報などを表示するアプリ。
- ユーザの地域、デバイス パラメータ、またはその他のユーザに応じたデータに基づいて、主たる機能が異なり、その違いについてストアの掲載情報でユーザに明確に宣伝していないアプリ。
- バージョン間で大幅に変更され、その変更についてユーザに（「最新情報」欄などで）通知をせず、ストアの掲載情報も更新しないアプリ。
- 審査時の動作を変更した、またはごまかしているアプリ。
- ダウンロードにコンテンツ配信ネットワーク（CDN）を利用しているのに、ダウンロードの前にユーザにメッセージを表示してダウンロード サイズを開示しないアプリ。

操作されたメディア

虚偽のまたは誤解を招くような情報や宣伝文句を画像、動画、テキストを通じて伝えることを助長する、またはその作成に役立つアプリは認められない。誤解を招く、または虚偽であることが明らかな画像、動画、テキストを助長、または固定化し、配慮が求められる事象、政治、社会問題など、社会的関心事に悪影響をもたらす可能性があると判断されたアプリは認められない。

メディアを操作または改変するアプリでは、明瞭さや品質の改善を目的とした慣習的で編集上許容される範囲の調整を超え、メディアが改変されていることを一般的な人が明確に識別できない可能性がある場合は、そのことを明示するか、改変したメディアの透かしを入れなければならない。ただし、公共性が高い場合や、風刺やパロディーであることが明らかな場合は例外として認められることがある。

違反の例

- 政治的に配慮が求められるイベント中のデモに著名人を登場させるアプリ。
- 配慮が求められるイベントに登場した著名人やメディアを利用して、アプリストアの掲載情報でメディア改変機能を宣伝するアプリ。
- メディアクリップを改変してニュース番組を模倣するアプリ。

参考資料

- Google Play デベロッパーポリシーセンター 虚偽の振る舞い

1.5.2.5 不実表示

以下のようなアプリやデベロッパー アカウントは許可されない。

- 他の人や組織になりすましたり、オーナーや主目的を偽装、隠ぺいしたりしている。
- ユーザに誤解を与えるような組織的行為に関与している。たとえば、配信元の国を偽装、隠ぺいしたり、コンテンツを別の国のユーザに配信したりするアプリやデベロッパー アカウントなどが該当する。
- 政治や社会問題、公衆の関心事に関連するコンテンツを扱っている場合に、他のアプリやサイト、デベロッパー、アカウントと連携して、デベロッパーやアプリの重要情報（身元など）を隠ぺい、偽装している。

参考資料

- Google Play デベロッパーポリシーセンター 不実表示

1.5.2.6 Google Play の対象 API レベルに関するポリシー

ユーザに安全で保護されたエクスペリエンスを提供するため、Google Play では**すべてのアプリ**に対し、以下の API レベルを対象とすることを義務付ける。

新規アプリとアプリアップデートは、Android の最新のメジャー バージョンのリリースから 1 年以内にその Android API レベルをターゲットにする必要がある。この要件を満たさない新規アプリとアプリ アップデートは、Google Play Console からのアプリの送信ができなくなる。

アップデートのない既存の Google Play アプリ: Android の最新のメジャー バージョンのリリースから 2 年を過ぎてもその Android API レベルをターゲットにしていないアプリは、それ以降のバージョンの Android OS を搭載したデバイスの新規ユーザからアクセスできなくなる。そのアプリを以前に Google Play からインストールしたユーザは、アプリでサポートされていればどのバージョンの Android OS でも、引き続きアプリを検索、再インストール、使用できる。

対象 API レベルの要件を満たす方法についての技術的なアドバイスについては、移行ガイドを確認すること。

具体的なスケジュールについては、こちらのヘルプセンター記事でご確認すること。

参考資料

- Google Play デベロッパーポリシーセンター Google Play の対象 API レベルに関するポリシー

1.5.3 知的財産権・知的所有権に関する Google Play での規則

アプリやデベロッパーアカウントが他者の知的所有権（商標権、著作権、特許権、企業秘密、その他の専有的権利を含む）を侵害する行為は認められない。知的所有権の侵害を助長または誘導するアプリも認められない。

Google では、著作権を侵害しているとする明確な通知を受けた場合には、それに対し適切に対応する。DMCA に基づく申し立てを行う方法など、詳しくは、著作権に関する手続きを確認すること。

アプリ内での偽造品の販売または宣伝について申し立てを行うには、偽造品の報告を行うこと。

Google Play のアプリがご自身の商標権を侵害していると思われる場合は、該当のデベロッパーに直接連絡して、問題の解決にあたることを推奨する。デベロッパーと問題を解決できない場合は、こちらのフォームから商標権侵害を申し立てすること。

アプリ内またはストアの掲載情報で第三者の知的財産（ブランド名、ロゴ、画像および映像など）を使用する許可を受けていることを証明する文書がある場合は、知的所有権の侵害を理由にアプリが否承認となることがないよう、アプリを送信する前に Google Play チームに連絡すること。

参考資料

- Google Play デベロッパーポリシーセンター 知的所有権

1.5.3.1 著作権で保護されているコンテンツの無断使用

著作権を侵害するアプリは認められない。著作権で保護されているコンテンツを改変することも違反となる場合がある。著作権で保護されているコンテンツを使用する場合は、その権利の証拠を示すよう求められることがある。

著作権で保護されているコンテンツをアプリ機能のデモとして使用する場合は、注意が必要である。オリジナルのものを作成することが、通常は最も安全な方法である。

違反の例

- 音楽のアルバム、ビデオゲーム、書籍のカバーアート
- 映画、テレビ、ビデオゲームのマーケティング画像
- マンガ、アニメ、映画、ミュージック ビデオ、テレビのアートワークや画像
- 大学やプロのスポーツチームのロゴ
- 有名人のソーシャル メディア アカウントから取得した写真
- 有名人のプロによる画像
- 著作権で保護されているオリジナル作品と区別がつきにくい複製または「ファンアート」
- 著作権で保護されているコンテンツの音声クリップを再生するサウンドボードを持つアプリ
- パブリック ドメイン以外の書籍の完全な複製や翻訳

1.5.3.2 著作権侵害の助長

著作権侵害を誘導または助長するアプリは認められない。アプリを公開する前に、著作権の侵害をアプリが助長することにならないかどうか確認し、必要に応じて法律上の助言を受けること。

違反の例

- 著作権で保護されているコンテンツのローカルコピーをユーザが許可なくダウンロードできるストリーミングアプリ。
- 音楽や動画などの著作権で保護されているコンテンツを、該当する著作権法に違反して、ユーザがストリーミングやダウンロードすることを助長するアプリ

1.5.3.3 商標権侵害

他者の商標を侵害するアプリは認められない。商標とは、商品やサービスの提供元を識別する語句、シンボル、またはその組み合わせです。商標権を取得した所有者には、特定の商品やサービスにその商標を使用する独占的な権利がある。

商標権の侵害とは、商品の提供元を混同させる可能性のある方法で、同一または類似の商標を不正にまたは無断で使用することである。こうした紛らわしい方法で他者の商標を使用するアプリは、公開が停止されることがある。

1.5.3.4 偽造品

偽造品の販売や宣伝を行うアプリは認められない。偽造品とは、他の商標と同一、またはほとんど区別がつかない商標やロゴを使用している商品を指す。このような商品は、真正品と偽って販売するために、対象商品のブランドの特徴を模倣したものである。

2 データストレージとプライバシー要件

2.1 MSTG-STORAGE-1

個人識別情報、ユーザ資格情報、暗号化鍵などの機密データを格納するために、システムの資格情報保存機能を使用している。

2.1.1 ハードウェア格納型 Android KeyStore

ハードウェア格納型 Android KeyStore は、Android の多層防御のセキュリティ概念に新たなレイヤーを提供する。Keymaster Hardware Abstraction Layer(HAL) は、Android 6 (API level 23) で導入された。

アプリケーションは、キーがセキュリティハードウェアの内部に保存されているかどうかを確認することができる (KeyInfo.isinsideSecureHardware が true を返すかどうかをチェックすることで確認できる)。Android 9 (API level 28) 以上のデバイスは StrongBoxKeymaster module を持つことができる。なお、KeyInfo.isinsideSecureHardware は API level 31 で廃止され、getSecurityLevel が推奨されている。

Android 9 (API level 28) 以降を実行しているデバイスは、StrongBox Keymaster モジュールを搭載できる。これは、独自の CPU 、Secure ストレージ、真の乱数ジェネレーター、パッケージ改ざんに対抗するメカニズムを持つハードウェアセキュリティモジュールに常駐する Keymaster HAL の実装である。この機能を使用するには、Android Keystore を使用してキーを生成またはインポートする際に、KeyGenParameterSpec.Builder クラスまたは KeyProtection.Builder クラスの setIsStrongBoxBacked メソッドに true を渡す必要がある。

実行時に StrongBox が使われるようにするには、isInsideSecureHardware (現在は非推奨) が true を返し、システムが StrongBoxUnavailableException を throw しないことを確認する。この Exception は StrongBox Keymaster が特定のアルゴリズムおよびキーに関連付けられたキーサイズで使用できない場合に throw される。ハードウェアベースの KeyStore の機能説明は、AOSP のページにある。

Keymaster HAL は、Android Keystore が使用する Trusted Execution Environment(TEE) や Secure Element (SE) といったハードウェアベースのコンポーネントへのインターフェースである。Titan M は、そのようなハードウェアを搭載したコンポーネントの一例である。

参考資料

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

ルールブック

- **キーがセキュリティハードウェアの内部に保存されているかどうかを確認する（推奨）**
- ***StrongBox* の利用方法（推奨）**

2.1.2 キー認証

暗号プリミティブによる多要素認証、クライアント側での機密データの安全な保存など、ビジネスに不可欠な操作のために Android Keystore に大きく依存するアプリケーションの場合、Android は Android Keystore を通じて管理される暗号材料のセキュリティを分析するのに役立つ Key Attestation の機能を提供する。Android 8.0 (API level 26) から、Google アプリの端末認証が必要な新しい端末（Android 7.0 以上）には、キーの認証が必須となった。このようなデバイスは、Google Hardware Attestation Root 証明書によって署名された認証キーが使用され、キー認証プロセスを通じて同じことを検証される。

キー認証の際、対称鍵のエイリアスを指定すると、その対称鍵のプロパティを検証するために使用できる証明書チェーンを得ることができる。このチェーンのルート証明書が Google Hardware Attestation Root 証明書で、ハードウェアへの対称鍵の保存に関するチェックが行われていれば、そのデバイスがハードウェアレベルのキー認証に対応しており、Google が安全であると考える hardware-backed keystore にキーがあることが保証される。あるいは、認証チェーンに他のルート証明書がある場合、Google はハードウェアのセキュリティについて主張しない。

キー認証プロセスはアプリケーション内に直接実装することもできるが、セキュリティ上の理由からサーバ側で実装することが推奨される。以下は、キー認証の安全な実装のためのハイレベルなガイドラインである。

- サーバは、CSPRNG（Cryptographically Secure Random Number Generator）を用いて乱数を安全に生成し、キー認証プロセスを開始する必要があり、同じものをチャレンジとしてユーザに送信する必要がある。
- クライアントは、サーバから受け取ったチャレンジで setAttestationChallenge API を呼び出し、KeyStore.getCertificateChain メソッドで証明書チェーンを取得する必要がある。
- 認証応答は検証のためにサーバに送信され、キー認証応答の検証のために以下のチェックが行われる必要がある。
  - ルートまでの証明書チェーンを検証し、有効性、完全性、信頼性などの証明書の sanity check を実行する。チェーン内の証明書がいずれも失効していないことを確認するために、Google が管理する証明書失効ステータスリストを確認する。
  - ルート証明書が、認証プロセスを信頼できるようにする Google 認証ルートキーで署名されているかどうかを確認する。
  - 証明書チェーンの最初の要素に表示される証明書拡張データを抽出し、以下のチェックを実行する。
    * 認証チャレンジが、認証プロセスを開始する際にサーバで生成されたものと同じ値であることを確認する。

    * キー認証応答で署名を確認する。
    * デバイスに安全なキーの保存メカニズムがあるかどうかを判断するために、Keymaster のセキュリティレベルを確認する。Keymaster はセキュリティコンテキストで動作するソフトウェアの一部であり、すべての安全なキーストア操作を提供する。セキュリティレベルは Software, TrustedEnvironment, StrongBox のいずれかになる。セキュリティレベルが TrustedEnvironment あるいは StrongBox で、かつ証明書チェーンに Google 証明書ルートキーで署名されたルート証明書が含まれている場合、クライアントはハードウェアレベルのキー認証に対応する。
    * クライアントのステータスを確認し、完全な信頼チェーン（検証済みのブートキー、ロックされたブートローダ、検証済みのブートステート）を確保する。
    * さらに、目的、アクセス時間、認証要件など、対称鍵の属性を確認することができる。

注意：何らかの理由でこのプロセスが失敗した場合、キーがセキュリティハードウェアにないことを意味する。これは、キーが危険にさらされていることを意味するものではない。

Android Keystore の認証応答の典型的な例は、以下のようになる。

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

↪"308202ca30820270a003020102020101300a06082a8648ce3d040302308188310b3009060355040613 0.
↪..",

↪"308202783082021ea003020102020210013 00a06082a8648ce3d040302308198310b30090603550406 1.
↪..",

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

上記の JSON スニペットにおいて、キーは以下の意味を持つ。

- fmt：認証文のフォーマット識別子
- authData：認証のための authenticator データを表す
- alg：署名に使用されるアルゴリズム
- sig：署名
- x5c：証明書チェーン

注：sig は authData と clientDataHash（サーバから送られたチャレンジ）を連結して生成し、credential な秘密

鍵を通して署名アルゴリズム（ alg ）を用いて署名し、同じものをサーバ側で最初の証明書の公開鍵を使って検証する。

実装ガイドラインの詳細については、Google のサンプルコードを参照すること。

セキュリティ解析の観点から、アナリストはキー認証の安全な実装のために、以下のチェックを行うこと。

- キー認証がクライアント側で完全に実装されているかどうかを確認する。このようなシナリオでは、アプリケーションを改ざんしたり、メソッドをフックしたりすることで、同じことを簡単に回避できる。
- キー認証を開始する際に、サーバがランダムチャレンジを使用しているかどうかを確認する。これを怠ると、安全でない実装となり、リプレイ攻撃に対して脆弱になる。また、チャレンジのランダム性に関してもチェックする必要がある。
- サーバがキー認証応答の完全性を検証しているかどうかを確認する。
- サーバがチェーン内の証明書に対して、完全性検証、信頼性検証、有効性などの基本的なチェックを実行しているかどうかを確認する。

参考資料

- owasp-mastg Data Storage Methods Overview Key Attestation

ルールブック

- **安全なキー認証をする場合は、サーバから受け取ったチャレンジによりデバイス内の証明書を提供する（推奨）**
- **セキュリティ解析の観点から、キー認証の安全な実装のためのチェック（必須）**

2.1.3 KeyStore へのセキュアキーのインポート

Android 9 (API level 28) では、Android Keystore にキーを安全にインポートする機能が追加された。Android Keystore は、まず PURPOSE_WRAP_KEY を使用して、認証証明書で保護する必要のある対称鍵を生成する。この対称鍵は、Android Keystore にインポートされるキーの保護を目的としている。暗号化されたキーは、インポートされたキーの使用方法の説明を含む SecureKeyWrapper 形式で ASN.1 エンコードされたメッセージとして生成される。その後、暗号化されたキーは、ラッピングキーを生成した特定のデバイスに属する Android Keystore ハードウェア内で復号されるため、デバイスのホストメモリに平文として表示されることはない。

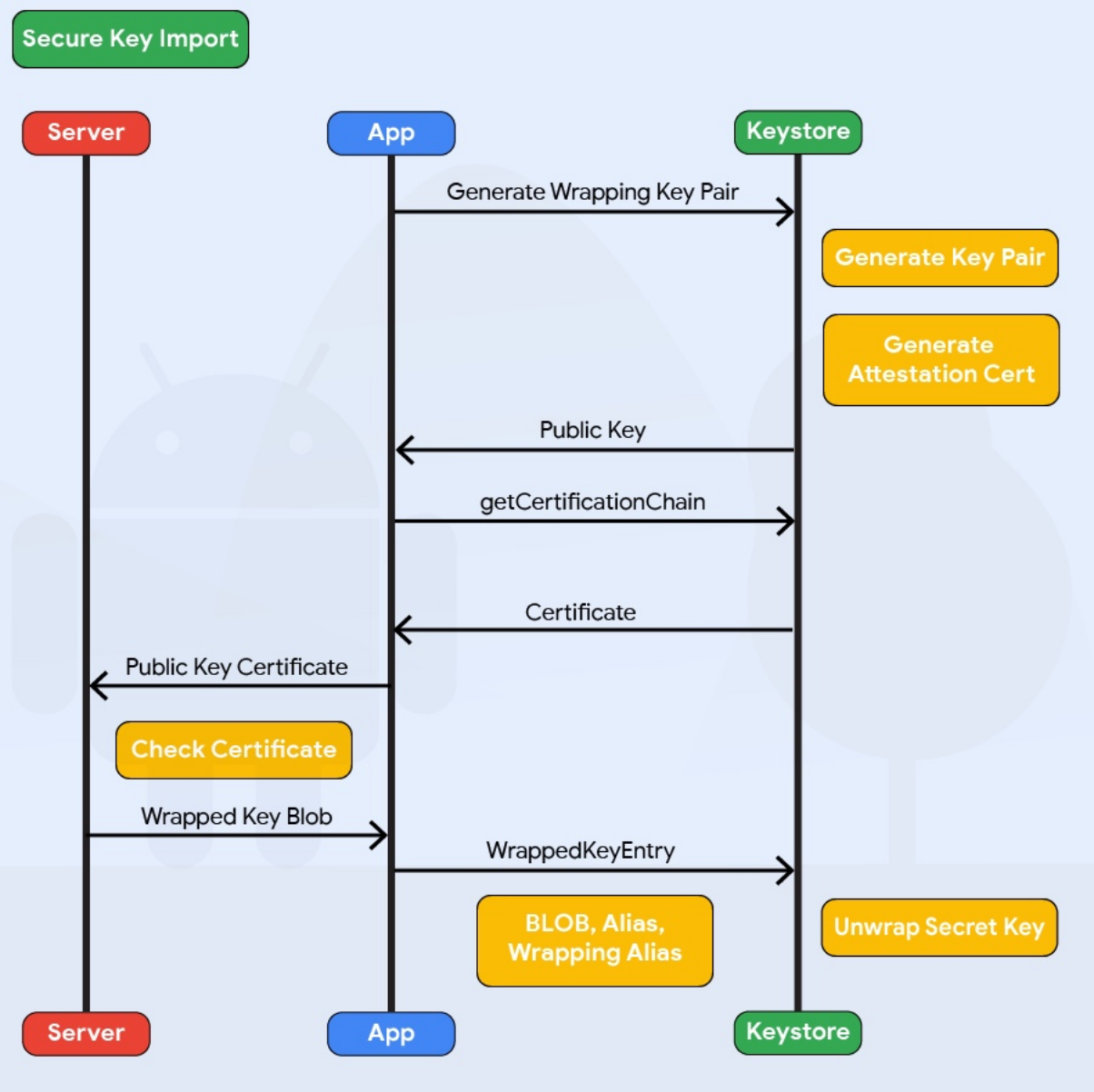


図 2.1.3.1　安全なキーのインポートフロー

Java での例:

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

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

上記のコードでは、SecureKeyWrapper 形式で暗号鍵を生成する際に設定する各種パラメータを示している。詳細は、Android の WrappedKeyEntry のドキュメントを参照。

KeyDescription AuthorizationList を定義する際、以下のパラメータが暗号化されたキーのセキュリティに影響を与える。

- algorithm : キーで使用する暗号化方式を指定
- keySize : キーのアルゴリズムの通常の方法で測定して、鍵のサイズをビット単位で指定
- digest : 署名および検証操作のためにキーとともに使用できるダイジェストアルゴリズムを指定

参考資料

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

2.1.4 古い KeyStore 実装

古いバージョンの Android には KeyStore は含まれていないが、JCA (Java Cryptography Architecture) の KeyStore インターフェースは含まれる。このインターフェースを実装した KeyStore を使用することで、KeyStore に保存されたキーの機密性と完全性を確保できる。BouncyCastle KeyStore (BKS) が推奨される。

すべての実装は、ファイルがファイルシステム上に保存されているという事実に基づく。すべてのファイルはパスワードで保護されている。作成するには、KeyStore.getInstance("BKS", "BC") メソッドを使用する。"BKS" は KeyStore 名 (BouncyCastle Keystore) で、"BC" は provider (BouncyCastle) を意味する。SpongyCastle をラッパーとして使用して、以下のように KeyStore を初期化することも可能となる。

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

すべての KeyStore が、KeyStore ファイルに保存されたキーを適切に保護するわけではないことに注意する必要がある。

参考資料

- owasp-mastg Data Storage Methods Overview Older KeyStore Implementations

ルールブック

- **古い *Android OS* では *BouncyCastle KeyStore* によりキーを保存する（推奨）**

2.1.5 Key Chain

Keychain クラスは、システム全体の秘密鍵とそれに対応する証明書 (チェーン) を保存および取得するために使用される。Keychain に何かを初めてインポートする場合、ユーザは証明書ストレージを保護するためにロック画面の PIN またはパスワードを設定するよう促される。Keychain はシステム全体であり、すべてのアプリケーションが Keychain に保存されている materials にアクセスできることに注意する必要がある。

ソースコードを調べて、Android のネイティブなメカニズムが機密情報を特定するかどうかを判断する。機密情報は暗号化し、平文で保存してはいけない。デバイスに保存する必要がある機密情報については、Keychain

クラスを介してデータを保護するために、いくつかの API 呼び出しを利用できる。以下のステップを完了する。

- アプリが Android KeyStore と Cipher のメカニズムを使用して、暗号化された情報をデバイスに安全に保存していることを確認する。AndroidKeystore, import java.security.KeyStore, import javax.crypto.Cipher, import java.security.SecureRandom, and corresponding usages というパターンを探してみる。
- store(OutputStream stream, char[] password) 関数を使用して、KeyStore をパスワード付きでディスクに保存する。パスワードはハードコードされたものではなく、ユーザによって提供されるものであることを確認する。

参考資料

- owasp-mastg Data Storage Methods Overview Keychain

ルールブック

- *Keychain* に初めてインポートする場合、ユーザへ証明書ストレージを保護するためにロック画面の *PIN* またはパスワードを設定するよう促す（必須）
- *Android* のネイティブなメカニズムが機密情報を特定するかどうかを判断する（必須）

2.1.6 暗号化キーの保存

Android KeyStore では、Android デバイス上でのキーの不正利用を防ぐため、キーの生成時やインポート時に、アプリが許可したキーの用途を指定できるようになっている。一度指定した内容は、変更することができない。

以下はキーの最も安全な保存方法から最も安全でない保存方法である。

- ハードウェアで格納された Android KeyStore にキーを保存する。
- 全てのキーをサーバに保存し、強力な認証の後に利用可能にする。
- マスターキーをサーバに保存し、Android の SharedPreferences に保存された他のキーを暗号化するために使用する。
- キーに十分な長さと Salt を持たせ、ユーザが提供する強力なパスフレーズから毎回導き出す。
- キーを Android KeyStore のソフトウェア実装に格納する。
- マスターキーを Android Keystore のソフトウェア実装に格納し、SharedPreferences に格納された他のキーを暗号化するために使用する。
- [非推奨] 全てのキーを SharedPreferences に保存する。
- [非推奨] キーをソースコードにハードコードする。
- [非推奨] 安定した属性に基づく予測可能な難読化関数または鍵導出関数を使用する。
- [非推奨] 生成されたキーを public な場所 (/sdcard/ など) に保存する。

参考資料

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

ルールブック

- **暗号化キーの保存方法（必須）**

2.1.6.1 ハードウェア格納型 Android KeyStore によるキーの保存

Android 7.0 (API level 24) 以上のデバイスで、利用可能なハードウェアコンポーネント (Trusted Execution Environment (TEE) または Secure Element (SE)) があれば、**ハードウェア格納型** *Android KeyStore* を使用できる。また、**安全なキー認証の実装のために提供されるガイドライン**を使用することで、キーがハードウェアで保護されていることを確認することができる。ハードウェアコンポーネントが利用できない場合や、Android 6.0 (API level 23) 以下のサポートが必要な場合は、キーをリモートサーバに保存し、認証後に利用できるようにすることが推奨される。

参考資料

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

2.1.6.2 サーバへの保存

キー管理サーバにキーを安全に保存することは可能だが、データを復号するにはアプリをオンラインにする必要がある。これは、特定のモバイル アプリのユース ケースでは制限となる可能性があり、アプリのアーキテクチャの一部となり、ユーザビリティに大きな影響を与える可能性があるため、慎重に検討する必要がある。

参考資料

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

2.1.6.3 ユーザ入力によるキーの導出

ユーザが入力したパスフレーズからキーを生成することは一般的な解決策だが（使用する Android API level によって異なる。）、ユーザビリティに影響し、攻撃対象に影響を与え、さらなる弱点をもたらす可能性がある。

アプリケーションが暗号化操作を行うたびに、ユーザのパスフレーズが必要になる。パスフレーズを毎回入力させるのは理想的なユーザエクスペリエンスとは言えない。パスフレーズはユーザが認証されている間、メモリに保持される。パスフレーズをメモリ内に保持することは、ベストプラクティスではない。キーをゼロにすることは、**「キーマテリアルの消去」**で説明したように、非常に困難な作業であることが多い。

さらに、パスフレーズから派生したキーには弱点があることを考慮する。例えば、パスワードやパスフレーズはユーザによって再利用されたり、簡単に推測されたりする可能性がある。詳しくは**「暗号のテスト」**の章を参照。

参考資料

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

2.1.6.4 キーマテリアルの消去

キーマテリアルは、不要になったらすぐにメモリから消去する必要がある。ガベージコレクタ（Java）や不変文字列（Kotlin）を使用する言語では、秘密データを実際にクリーンアップするには一定の限界がある。Java Cryptography Architecture Reference Guide では、機密データを格納するために String の代わりに char[] を使用し、使用後に配列を null にすることを提案する。

一部の暗号はバイト配列のクリーンアップを適切に行わないことに注意する。例えば、BouncyCastle の AES 暗号は、常に最新の作業キーをクリーンアップするわけではなく、メモリ上にバイト配列のコピーをいくつか残している。次に、BigInteger ベースのキー（例えば秘密鍵）は、追加の労力なしにヒープから削除したり、ゼロにしたりすることはできない。バイト配列のクリアは、Destroyable を実装したラッパーを作成することで実現できる。

参考資料

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

ルールブック

- **キーマテリアルは、不要になったらすぐにメモリから消去する必要がある（必須）**

2.1.6.5 Android KeyStore API を使用したキーの保存

よりユーザフレンドリーで推奨される方法は、Android KeyStore API システム（それ自体または Keychain を経由して）を使用してキーマテリアルを保存することである。可能であれば、ハードウェア格納型ストレージを使用すべきである。そうでない場合は、Android KeyStore のソフトウェア実装にフォールバックする必要がある。ただし、Android KeyStore API は、Android のさまざまなバージョンで大幅に変更されていることに注意が必要である。

以前のバージョンでは、Android KeyStore API は公開鍵/秘密鍵ペア（例：RSA ）の保存のみをサポートしている。対称鍵のサポートは、Android 6.0（API level 23）以降に追加された。そのため、開発者は、対称鍵を安全に保存するために、さまざまな Android API level を扱う必要がある。

参考資料

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

ルールブック

- **キーマテリアルを保存する（推奨）**

2.1.6.6 キーを他のキーで暗号化して保存する

Android 5.1（API level 22）以下のデバイスで対称鍵を安全に保存するには、公開鍵と秘密鍵のペアを生成する必要がある。公開鍵を用いて共通鍵を暗号化し、秘密鍵を Android KeyStore に保存する。暗号化された共通鍵は、base64 でエンコードして SharedPreferences に格納することが可能となる。対称鍵が必要なときはいつでも、アプリケーションが Android KeyStore から秘密鍵を取り出し、対称鍵を復号する。

エンベロープ暗号化またはキーラッピングは、共通鍵暗号方式を使用してキー マテリアルをカプセル化する同様のアプローチである。Data encryption keys(DEKs) は、安全に保管されている key encryption key (KEKs) で暗号化できる。暗号化された DEKs は、SharedPreferences に保存するか、ファイルに書き込むことができる。必要に応じて、アプリケーションは KEK を読み取り、DEK を復号する。暗号化キーの暗号化の詳細については、OWASP Cryptographic Storage Cheat Sheet を参照。

また、このアプローチの説明として、androidx.security.crypto パッケージの EncryptedSharedPreferences を参照。

参考資料

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

ルールブック

- *Android OS 5.1* 以下で対称鍵を安全に保管する場合は公開鍵と秘密鍵のペアを生成する（必須）
- *EncryptedSharedPreferences* の利用方法（推奨）

2.1.6.7 キーを保存するための安全でないオプション

暗号化キーの保存方法として、Android の SharedPreferences に保存する方法があるが、これはあまり安全ではない。SharedPreferences を使用する場合、ファイルを作成したアプリケーションのみがファイルを読み取ることができる。しかし、root 化されたデバイスでは、ルートアクセス権を持つ他のアプリケーションは、他のアプリケーションの SharedPreferences ファイルを簡単に読み取ることができる。Android KeyStore には当てはまらない。Android KeyStore のアクセスはカーネルレベルで管理されているため、Android KeyStore がキーを消去したり破壊したりせずにバイパスするには、かなり多くの作業とスキルが必要となる。

最後の 3 つのオプションは、ソースコードにハードコードされた暗号鍵を使用すること、安定した属性に基づく予測可能な難読化機能または鍵導出関数機能を持つこと、生成したキーを /sdcard/ などの public の場所に格納することである。ハードコードされた暗号化キーは、アプリケーションのすべてのインスタンスが同じ暗号化キーを使用することを意味するため、問題となる。攻撃者は、アプリケーションのローカルコピーをリバースエンジニアリングして暗号鍵を取り出し、そのキーを使って、どのデバイス上でもアプリケーションによって暗号化されたデータを復号することができる。

次に、他のアプリケーションからアクセス可能な識別子に基づく予測可能な鍵導出関数がある場合、攻撃者は KDF を見つけてデバイスに適用するだけでキーを見つけることができる。最後に、他のアプリケーションが public パーティションを読み取る権限を持ち、キーを盗むことができるため、暗号化キーを public に保存することも強く推奨しない。

参考資料

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

ルールブック

- 安全性の低い暗号化キーの保存方法は利用しない（必須）

2.1.7 サードパーティライブラリ

Android プラットフォームに特化した暗号化機能を提供するオープンソースのライブラリがいくつか存在する。

- Java AES Crypto - 文字列を暗号化・復号するためのシンプルな Android クラス。
- SQL Cipher - SQLCipher は、SQLite のオープンソース拡張機能で、データベースファイルの透過的な 256 ビット AES 暗号化を提供する。
- Secure Preferences - Android Shared preference wrapper は Shared Preferences の keys と values の暗号化を提供する。
- Themis - 認証、ストレージ、メッセージングなどのデータを保護するために、多くのプラットフォームで同じ API を提供するクロスプラットフォームの高レベル暗号化ライブラリ。

キーが KeyStore に保存されていない限り、root 化されたデバイスでキーを簡単に取得し、保護しようとしている値を復号することが常に可能であることを念頭におく必要がある。

参考資料

- owasp-mastg Data Storage Methods Overview Third Party libraries

ルールブック

- サードパーティライブラリでの暗号化（非推奨）

2.1.8 ルールブック

1. キーがセキュリティハードウェアの内部に保存されているかどうかを確認する（推奨）
2. *StrongBox* の利用方法（推奨）
3. 安全なキー認証をする場合は、サーバから受け取ったチャレンジによりデバイス内の証明書を提供する（推奨）
4. セキュリティ解析の観点から、キー認証の安全な実装のためのチェック（必須）
5. 古い *Android OS* では *BouncyCastle KeyStore* によりキーを保存する（推奨）
6. *Keychain* に初めてインポートする場合、ユーザへ証明書ストレージを保護するためにロック画面の *PIN* またはパスワードを設定するよう促す（必須）
7. *Android* のネイティブなメカニズムが機密情報を特定するかどうかを判断する（必須）
8. 暗号化キーの保存方法（必須）

9. キーマテリアルは、不要になったらすぐにメモリから消去する必要がある（必須）
10. キーマテリアルを保存する（推奨）
11. *Android OS 5.1* 以下で対称鍵を安全に保管する場合は公開鍵と秘密鍵のペアを生成する（必須）
12. *EncryptedSharedPreferences* の利用方法（推奨）
13. 安全性の低い暗号化キーの保存方法は利用しない（必須）
14. サードパーティライブラリでの暗号化（非推奨）

2.1.8.1 キーがセキュリティハードウェアの内部に保存されているかどうかを確認する（推奨）

キーがセキュリティハードウェアの内部に保存されているかどうかを確認することができる(KeyInfo.isinsideSecureHardware が true を返すかどうかをチェックすることで確認できる)。

確認方法は以下である。API level 31 以降は、isinsideSecureHardware は非推奨であるため API level 31 以降をターゲットとするアプリは getSecurityLevel を使用する必要がある。

```
      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();
      }
```

これに注意しない場合、以下の可能性がある。

- キーがユーザにより読み書きされ、悪用される可能性がある。

2.1.8.2 StrongBox の利用方法（推奨）

Android 9 (API level 28) 以降を実行しているデバイスは、StrongBox Keymaster モジュールを搭載できる。これは、独自の CPU 、Secure ストレージ、真の乱数ジェネレーター、パッケージ改ざんに対抗するメカニズムを持つハードウェアセキュリティモジュールに常駐する Keymaster HAL の実装である。この機能を使用するには、Android Keystore を使用してキーを生成またはインポートする際に、KeyGenParameterSpec.Builder クラスまたは KeyProtection.Builder クラスの setIsStrongBoxBacked メソッドに true を渡す必要がある。実行時に StrongBox が使われるようにするには、isInsideSecureHardware が true を返し、システムが StrongBoxUnavailableException を throw しないことを確認する。なお、isinsideSecureHardware は API level 31 で廃止され、getSecurityLevel が推奨されている。

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

これに注意しない場合、以下の可能性がある。

- キーがユーザにより読み書きされ、悪用される可能性がある。

2.1.8.3 安全なキー認証をする場合は、サーバから受け取ったチャレンジによりデバイス内の証明書を提供する（推奨）

キー認証はクライアント側のみで実装可能であるが、より安全な認証を行うためにはサーバ側で実装することが推奨される。その場合クライアント側では、サーバから受け取ったチャレンジで setAttestationChallenge API を呼び出し、KeyStore.getCertificateChain メソッドで証明書チェーンを取得し、証明書をサーバへ提供する必要がある。サーバは提供された証明書により、キー認証を行う。

以下へ上記処理のサンプルコードを示す。

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

これに注意しない場合、以下の可能性がある。

- チャレンジを受け取ったデバイスが該当デバイスであることを証明できず、安全なキー認証を出来ない可能性がある。

2.1.8.4 セキュリティ解析の観点から、キー認証の安全な実装のためのチェック（必須）

セキュリティ解析の観点から、アナリストはキー認証の安全な実装のために、以下のチェックを行うこと。

- キー認証がクライアント側で完全に実装されていないことを確認する。完全に実装されている場合、アプリケーションを改ざんしたり、メソッドをフックしたりすることで、同じことを簡単に回避できてしまう。
- キー認証を開始する際に、サーバがランダムチャレンジを使用していることを確認する。これを怠ると、安全でない実装となり、リプレイ攻撃に対して脆弱になる。また、チャレンジのランダム性に関してもチェックする必要がある。

- サーバがキー認証応答の完全性を検証していることを確認し、検証していない場合は安全でない実装となるため対応が必要である。
- サーバがチェーン内の証明書に対して、完全性検証、信頼性検証、有効性などの基本的なチェックを実行していることを確認し、チェックしていない場合は安全でない実装となるため対応が必要である。

これに注意しない場合、以下の可能性がある。

- 安全なキー認証を保証できない。

2.1.8.5 古い Android OS では BouncyCastle KeyStore によりキーを保存する（推奨）

古いバージョンの Android には KeyStore は含まれていないが、JCA (Java Cryptography Architecture) の KeyStore インターフェースは含まれる。このインターフェースを実装した KeyStore を使用することで、KeyStore に保存されたキーの機密性と完全性を確保できる。その中で、BouncyCastle KeyStore (BKS) が推奨される。

BouncyCastle KeyStore (BKS) を使用した KeyStore の実装を以下に記載する。"BKS" は KeyStore 名 (BouncyCastle Keystore) で、"BC" は provider (BouncyCastle) を意味する。SpongyCastle をラッパーとして使用して、以下のように KeyStore を初期化することも可能となる。

すべての KeyStore が、KeyStore ファイルに保存されたキーを適切に保護するわけではないことに注意する必要がある。

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

これに注意しない場合、以下の可能性がある。

- 使用するキーの機密性と完全性を確保できない可能性がある。

2.1.8.6 Keychain に初めてインポートする場合、ユーザへ証明書ストレージを保護するためにロック画面の PIN またはパスワードを設定するよう促す（必須）

Keychain に何かを初めてインポートする場合、ユーザへ証明書ストレージを保護するためにロック画面の PIN またはパスワードを設定するよう促す。Keychain はシステム全体であり、すべてのアプリケーションが Keychain に保存されている materials にアクセスできることに注意する必要がある。

これに違反する場合、以下の可能性がある。

- Keychain にインポートされた情報はシステムレベルで利用できてしまうため、第三者にデバイスを使用されると意図しない用途で使用される可能性がある。

2.1.8.7 Android のネイティブなメカニズムが機密情報を特定するかどうかを判断する（必須）

ソースコードを調べて、Android のネイティブなメカニズムが機密情報を特定するかどうかを判断する。機密情報は暗号化し、平文で保存してはいけない。デバイスに保存する必要がある機密情報については、Keychain クラスを介してデータを保護するために、いくつかの API 呼び出しを利用できる。以下のステップを完了する。

- アプリが暗号化された情報をデバイスに保存しているか確認する。
- アプリが Android KeyStore と Cipher のメカニズムを使用して、暗号化された情報をデバイスに安全に保存していることを確認する。

  以下のパターンを探してみる。

  - AndroidKeystore
  - import java.security.KeyStore
  - import javax.crypto.Cipher
  - import java.security.SecureRandom, and corresponding usages
- store(OutputStream stream, char[] password) 関数を使用して、KeyStore をパスワード付きでディスクに保存する。パスワードはハードコードされたものではなく、ユーザによって提供されるものであることを確認する。以下へサンプルコードを示す。

```
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();
}
```

Keychain クラスでの資格情報のインストール方法を以下サンプルコードへ示す。

```
    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()
```

(次のページに続く)

(前のページからの続き)

```
        installIntent.putExtra(Keychain.EXTRA_PKCS12, keychain)
        installIntent.putExtra(Keychain.EXTRA_NAME, /*alias*/)
        val intent = Intent(this, /*Activity name*/::class.java)
        launcher.launch(intent)
    }
```

これに違反する場合、以下の可能性がある。

- 機密情報が平文で保存され、第三者に漏洩する可能性がある。

2.1.8.8 暗号化キーの保存方法（必須）

暗号化キーの安全な保存方法、安全でない保存方法は以下である。

推奨

以下にキーの推奨される保存方法を安全な順に記載する。

- ハードウェアで格納された Android KeyStore にキーを保存する。

  以下は Android KeyStore にキーを保存するサンプルコード。

```
      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();
      }
```

- 全てのキーをサーバに保存し、強力な認証の後に利用可能にする。※詳細・注意事項については「**サーバへの保存**」を参照。
- マスターキーをサーバに保存し、Android の SharedPreferences に保存された他のキーを暗号化するために使用する。※詳細・注意事項については「**サーバへの保存**」を参照。
- キーに十分な長さと Salt を持たせ、ユーザが提供する強力なパスフレーズから毎回導き出す。※詳細・注意事項については「**ユーザ入力によるキーの導出**」を参照。
- キーを Android KeyStore のソフトウェア実装に格納する。※詳細・注意事項については「**キーマテリアルの消去**」と「*Android KeyStore API* **を使用したキーの保存**」を参照。
- マスターキーを Android Keystore のソフトウェア実装に格納し、SharedPreferences に格納された他のキーを暗号化するために使用する。※詳細・注意事項については「**キーマテリアルの消去**」と「*Android KeyStore API* **を使用したキーの保存**」を参照。

非推奨

- 全てのキーを SharedPreferences に保存する。
- キーをソースコードにハードコードする。
- 安定した属性に基づく予測可能な難読化関数または鍵導出関数を使用する。
- 生成されたキーを public な場所 (/sdcard/ など) に保存する。

これに違反する場合、以下の可能性がある。

- Android デバイス上でのキーを不正利用される。

2.1.8.9 キーマテリアルは、不要になったらすぐにメモリから消去する必要がある（必須）

キーマテリアルは、不要になったらすぐにメモリから消去する必要がある。ガベージコレクタ（Java）や不変文字列（Kotlin）を使用する言語では、秘密データを実際にクリーンアップするには一定の限界がある。Java Cryptography Architecture Reference Guide では、機密データを格納するために String の代わりに char[] を使用し、使用後に配列を null にすることを提案する。

Java での例:

```
        // 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);
```

(次のページに続く)

(前のページからの続き)

```
        cleartext = null;
        ciphertext = null;
```

一部の暗号はバイト配列のクリーンアップを適切に行わないことに注意する。例えば、BouncyCastle の AES 暗号は、常に最新の作業キーをクリーンアップするわけではなく、メモリ上にバイト配列のコピーをいくつか残している。次に、BigInteger ベースのキー（例えば秘密鍵）は、追加の労力なしにヒープから削除したり、ゼロにしたりすることはできない。バイト配列のクリアは、Destroyable を実装したラッパーを作成することで実現できる。

Java での例:

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

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

これに違反する場合、以下の可能性がある。

- メモリ内のキーマテリアルが、別の用途で使用される可能性がある。
- ガベージコレクタ（Java）や不変文字列（Kotlin）を使用する言語ではクリーンアップされない可能性がある。

2.1.8.10 キーマテリアルを保存する（推奨）

よりユーザフレンドリーで推奨される方法は、Android KeyStore API システム（それ自体または Keychain を経由して）を使用してキーマテリアルを保存することである。

KeyStore API を使用する保存方法は以下となる。

Java での例:

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

これに注意しない場合、以下の可能性がある。

- キーを安全に保存できず、悪用される可能性がある。

2.1.8.11 Android OS 5.1 以下で対称鍵を安全に保管する場合は公開鍵と秘密鍵のペアを生成する（必須）

Android 5.1（API level 22）以下のデバイスで対称鍵を安全に保存するには、公開鍵と秘密鍵のペアを生成する必要がある。公開鍵を用いて共通鍵を暗号化し、秘密鍵を Android KeyStore に保存する。暗号化された共通鍵は、base64 でエンコードして SharedPreferences に格納することが可能となる。対称鍵が必要なときはいつでも、アプリケーションが Android KeyStore から秘密鍵を取り出し、対称鍵を復号する。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- 対称鍵を安全に保存できず、悪用される可能性がある。

2.1.8.12 EncryptedSharedPreferences の利用方法（推奨）

キーを他のキーで暗号化を行う場合の androidx.security.crypto パッケージの EncryptedSharedPreferences の対応方法は以下となる。

Java での例:

```
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();
```

これに違反する場合、以下の可能性がある。

- 対称鍵を安全に保存できず、悪用される可能性がある。

2.1.8.13 安全性の低い暗号化キーの保存方法は利用しない（必須）

安全性の低い暗号化キーの保存方法とされる SharedPreferences やハードコードなどの利用は危険であるため利用しない。

暗号化キーを格納する安全性の低い方法は、以下である。

- SharedPreferences への格納。root 化されたデバイスでは、ルートアクセス権を持つ他のアプリケーションは、他のアプリケーションの SharedPreferences ファイルを簡単に読み取ることができる。
- ソースコードにハードコードする。攻撃者は、アプリケーションのローカルコピーをリバースエンジニアリングして暗号鍵を取り出し、そのキーを使って、どのデバイス上でもアプリケーションによって暗

号化されたデータを復号することができる

- 他のアプリケーションからアクセス可能な識別子に基づく予測可能な鍵導出関数がある場合、攻撃者は KDF を見つけてデバイスに適用するだけでキーを見つけることができる。
- 暗号化キーを public に保存する。他のアプリケーションが public パーティションを読み取る権限を持ち、キーを盗むことができるため推奨しない。

これに違反する場合、以下の可能性がある。

- root 化されたデバイスの場合、SharedPreferences ファイルの暗号化キーを他のアプリケーションから読み取られる。
- リバースエンジニアリングにより、ハードコードされた暗号化キーを読み取られる。
- 他のアプリケーションからアクセス可能な識別子に基づく予測可能な鍵導出関数がある場合、攻撃者は KDF を見つけてデバイスに適用するだけでキーを見つけることができる。
- 暗号化キーを public に保存すると、他のアプリケーションが public パーティションを読み取る権限を持つ場合に、キーを盗むことができる。

2.1.8.14 サードパーティライブラリでの暗号化（非推奨）

Android プラットフォームに特化した暗号化機能を提供するオープンソースのライブラリとして、以下のライブラリが存在する。ライブラリは便利であるが、キーが KeyStore に保存されていない限り、root 化されたデバイスではキーを簡単に取得でき、保護しようとしている値を復号することが常に可能であるため、利用は推奨されない。

- Java AES Crypto：文字列を暗号化および復号するためのシンプルな Android クラス。
- SQL Cipher：SQLCipher は、SQLite のオープンソース拡張機能で、データベースファイルの透過的な 256 ビット AES 暗号化を提供する。
- Secure Preferences：Android Shared preference wrapper は Shared Preferences の keys と values の暗号化を提供する。
- Themis：認証、ストレージ、メッセージングなどのデータを保護するために、多くのプラットフォームで同じ API を提供するクロスプラットフォームの高レベル暗号化ライブラリ。

これが非推奨である理由は以下である。

- キーが KeyStore に保存されていない限り、root 化されたデバイスではキーを簡単に取得でき、保護しようとしている値を復号することが常に可能である。

2.2 MSTG-STORAGE-2

機密データはアプリコンテナまたはシステムの資格情報保存機能の外部に保存されていない。

2.2.1 内部ストレージ

デバイスの内部ストレージにファイルを保存することができる。内部ストレージに保存されたファイルは、デフォルトでコンテナ化され、デバイス上の他のアプリからアクセスすることはできない。ユーザがアプリをアンインストールすると、これらのファイルは削除される。以下のコードでは、機密データを内部ストレージに永続的に保存する。

Java での例:

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

Kotlin での例:

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

ファイルモードを確認し、アプリだけがファイルへアクセスできるようにする必要がある。このアクセスは MODE_PRIVATE で設定することができる。MODE_WORLD_READABLE（非推奨）や MODE_WORLD_WRITEABLE（非推奨）などのモードは、セキュリティ上のリスクをもたらす可能性がある。

FileInputStream クラスを検索して、アプリ内でどのファイルが開かれ、読み取られるかを見つける。

参考資料

- owasp-mastg Data Storage Methods Overview Internal Storage

ルールブック

- **機密データはアプリコンテナまたはシステムの資格情報保存機能へ保存する（必須）**

2.2.2 外部ストレージ

すべての Android 互換デバイスは、共有の外部ストレージをサポートしている。このストレージは、リムーバブル（ SD カードなど）または内蔵（非リムーバブル）である。外部ストレージに保存されたファイルは誰でも読み取り可能である。USB 大容量ストレージが有効な場合、ユーザはそれらを変更することができる。以下のコードを使用することで、機密情報を password.txt の内容として外部ストレージに永続的に保存することができる。

Java での例:

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

Kotlin での例:

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

アクティビティが呼び出されると、ファイルが作成されデータが外部ストレージの平文ファイルに保存される。

また、ユーザがアプリケーションをアンインストールしても、アプリケーションフォルダ（ data/data/<package-name>/ ）の外側に保存されたファイルは削除されないことも理解しておく必要がある。最後に、攻撃者が外部ストレージを使用して、場合によってはアプリケーションを任意に制御できることに注意する必要がある。詳細については、Checkpoint 社のブログを参照する。

参考資料

- owasp-mastg Data Storage Methods Overview External Storage

2.2.3 SharedPreferences

SharedPreferences API は、通常、キーと値のペアの小さなコレクションを永続的に保存するために使用される。SharedPreferences オブジェクトに格納されたデータは、プレーンテキストの XML ファイルに書き込まれる。SharedPreferences オブジェクトは、誰でも読み取り可能 (すべてのアプリからアクセス可能) または非公開として宣言できる。SharedPreferences API を誤って使用すると、機密データが流出する可能性がある。利用する場合は次の例を参考に検討する必要がある。

Java での例:

```
SharedPreferences sharedPref = getSharedPreferences("key", MODE_WORLD_READABLE);
SharedPreferences.Editor editor = sharedPref.edit();
```

(次のページに続く)

(前のページからの続き)

```
editor.putString("username", "administrator");
editor.putString("password", "supersecret");
editor.commit();
```

Kotlin での例:

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

アクティビティが呼び出されると、提供されたデータを使用して key.xml ファイルが作成される。このコードは、いくつかのベストプラクティスに違反している。

- ユーザ名とパスワードは平文で /data/data/<package-name>/shared_prefs/key.xml に保存される。

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

- MODE_WORLD_READABLE は、すべてのアプリケーションが key.xml のコンテンツにアクセスして読み取ることを許可する。

```
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
```

※ MODE_WORLD_READABLE と MODE_WORLD_WRITEABLE は、API level 17 以降では非推奨になっていることに注意する。新しいデバイスはこの影響を受けない可能性があるが、android:targetSdkVersion の値が 17 未満でコンパイルされたアプリケーションは、Android 4.2（API level 17）より前にリリースされた OS バージョンで実行される場合、影響を受ける可能性がある。

参考資料

- owasp-mastg Data Storage Methods Overview Shared Preferences

2.2.4 データベース

Android プラットフォームは、前のリストで前述したように、多数のデータベースオプションを提供する。各データベースオプションには、理解する必要のある独自の癖やメソッドが存在する。

参考資料

- owasp-mastg Data Storage Methods Overview Databases

2.2.4.1 SQLite

SQLite データベース（非暗号化）SQLite は、.db ファイルにデータを格納する SQL データベースエンジンである。Android SDK には、SQLite データベースのサポートが組み込まれている。データベースの管理に使用される主なパッケージは android.database.sqlite である。例えば、次のコードを使用して、機密情報を Activity 内に格納できる。

Java での例：

```
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();
```

Kotlin での例：

```
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()
```

Activity が呼び出されると、提供されたデータを使用してデータベースファイル privateNotSoSecure が作成され、平文ファイル /data/data/<package-name>/databases/privateNotSoSecure に保存される。

データベースのディレクトリには、SQLite データベース以外にいくつかのファイルが含まれる場合がある。

- Journal files：アトミックコミットとロールバックを実装するために使用される一時ファイル。
- Lock files：ロックおよびジャーナリング機能の一部で、SQLite の同時実行性を向上させ、writer 不足の問題を軽減するように設計されている。

機密情報は、暗号化されていない SQLite データベースに保存しないこと。

SQLite データベース（暗号化）ライブラリ SQLCipher を使用すると、SQLite データベースをパスワードで暗号化できる。

Java での例：

```
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();
```

Kotlin での例：

```
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()
```

データベースキーを安全に取得するには、次の方法がある。

- アプリを起動すると、PIN またはパスワードを使用してデータベースを復号するようにユーザに要求する（脆弱なパスワードと PIN はブルートフォース攻撃に対して脆弱である）。
- キーをサーバに保存し、Web サービスからのみアクセスできるようにする（デバイスがオンラインの場合にのみアプリを使用できるようにする）。

参考資料

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

2.2.4.2 Firebase

Firebase は 15 を超える製品を備えた開発プラットフォームであり、その内の 1 つが Firebase Real-time Database である。アプリケーション開発者が活用することで、NoSQL のクラウドホスティングデータベースにデータを保存し、同期させることができる。データは JSON として保存され、接続されているすべてのクライアントとリアルタイムで同期し、アプリケーションがオフラインになっても引き続き利用できる。

誤って構成された Firebase インスタンス、は次のネットワーク呼び出しを行うことで識別できる。

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

firebaseProjectName は、アプリケーションをリバースエンジニアリングすることにより、モバイルアプリケーションから取得できる。または、アナリストは以下に示すように、上記のタスクを自動化する Python スクリプトである Firebase Scanner を使用できる。

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

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

参考資料

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

2.2.4.3 Realm

Realm Database for Java は、開発者の間でますます人気が高まっている。データベースとそのコンテンツは、構成ファイルに格納されているキーを使用して暗号化できる。

```
//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);
```

データベースが暗号化されていない場合は、データを取得できてしまうため確認が必要。データベースが暗号化されている場合は、キーがソースまたはリソースにハードコードされているかどうか、共有設定またはその他の場所に保護されずに保存されているかどうかを確認すること。

参考資料

- owasp-mastg Data Storage Methods Overview Realm Databases

2.2.5 ルールブック

1. 機密データはアプリコンテナまたはシステムの資格情報保存機能へ保存する（必須）

2.2.5.1 機密データはアプリコンテナまたはシステムの資格情報保存機能へ保存する（必須）

機密データを安全に保存するためには、システムの資格情報保存機能（ Android Keystore ）で管理されたキーによるデータの暗号化と、暗号化されたデータをアプリコンテナ (内部ストレージ) へ保存を徹底する必要がある。Android Keystore の利用方法については、「暗号化キーの保存方法（必須）」を参照。

また、外部からの読み取りを避けるために、アプリだけが内部ストレージ内のファイルを読み書きできるように実装する必要がある。内部ストレージ内のファイルを作成する方法として、ストリームを使用する方法がある。この方法では、Context#openFileOutput を呼び出して、filesDir ディレクトリ内のファイルへアクセスするための FileOutputStream オブジェクトを取得する。指定のファイルが存在しない場合は、新規でファイルが作成される。

Context#openFileOutput の呼び出しでは、ファイルモードを指定する必要がある。指定するファイルモードによって、作成されるファイルの読み書き可能な範囲が決まる。下記は主なファイルモードである。

- MODE_PRIVATE
- MODE_WORLD_READABLE
- MODE_WORLD_WRITEABLE

下記は Context#openFileOutput 呼び出しの一例。なお、Android 7.0（ API level 24 ）以降を搭載したデバイスでは、ファイルモードに MODE_PRIVATE を指定しない場合、呼び出し時に SecurityException が発生する。

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

ファイルモード設定 MODE_PRIVATE デフォルトのモードで、作成されたファイルは呼び出し元のアプリケーション、又は同じユーザ ID を共有するすべてのアプリケーションがアクセスできる。

```
public static final int MODE_PRIVATE
```

ファイルモード設定 MODE_WORLD_READABLE 作成されたファイルへの読み取りアクセスを、他のすべてのアプリケーションが可能となる。なお、MODE_WORLD_READABLE の使用は API level 17 以降非推奨となっている。

```
public static final int MODE_WORLD_READABLE
```

ファイルモード設定 MODE_WORLD_WRITEABLE 作成されたファイルへの読み取りアクセスを、他のすべてのアプリケーションが可能となる。なお、MODE_WORLD_WRITEABLE の使用は API level 17 以降非推奨となっている。

```
public static final int MODE_WORLD_WRITEABLE
```

これに違反する場合、以下の可能性がある。

- 他のアプリや第三者に機密データを読み取られる。

2.3 MSTG-STORAGE-3

機密データはアプリケーションログに書き込まれていない。

2.3.1 ログ出力

このテストケースでは、システムログとアプリケーションログの両方から、機密性の高いアプリケーションデータを特定することに重点を置いている。以下のチェックを実施すること。

- ソースコードを解析し、ロギングに関連するコードを確認する。
- アプリケーションデータディレクトリにログファイルがあるかどうか確認する。
- システムメッセージとログを収集し、機密データが含まれていないかを分析する。

機密性の高いアプリケーションデータの漏洩を防ぐための一般的な推奨事項として、アプリケーションで必要と見なされる又はセキュリティ監査の結果などで安全であると明示されない限り、ログ記録は本番リリースから削除するべきである。

参考資料

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

2.3.1.1 ファイル書き込み

アプリケーションは、ログを作成するために Log クラスと Logger クラスを使用する。これを見つけるには、アプリケーションのソースコードを検証し、そのようなロギングクラスがないかを調べる。これらは多くの場合、以下のキーワードで検索することで見つけることができる。

- 関数/クラスでのキーワード
  - android.util.Log
  - Log.d | Log.e | Log.i | Log.v | Log.w | Log.wtf
  - Logger
- システム出力に関するキーワード
  - System.out.print | System.err.print
  - logfile
  - logging
  - logs

参考資料

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

ルールブック

- **ログを出力する場合は出力内容に機密情報を含めない（必須）**

2.3.1.2 Logcat 出力

モバイルアプリケーションのすべての機能を少なくとも一度は使用し、アプリケーションのデータディレクトリを特定し、ログファイル（/data/data/<パッケージ名>）を探す。アプリケーションのログを確認し、ログデータが生成されているかどうかを判断する。一部のモバイルアプリケーションでは、データディレクトリに独自のログを作成し保存している。

多くのアプリケーション開発者は、適切なロギングクラスの代わりに System.out.println または printStackTrace を使用している。したがって、テスト戦略にはアプリケーションの起動、実行、終了時に生成される全ての出力を含める必要がある。System.out.println または printStackTrace によって直接出力されるデータを特定する場合は Logcat を使用する。

以下のように Logcat の出力をフィルタリングすることで、特定のアプリをターゲットにすることができる。

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

※アプリの PID が既にわかっている場合は、--pid フラグを使用して直接指定が可能。

ログに特定の文字列またはパターンが表示される場合は、さらにフィルタまたは正規表現を適用することも可能である（例. Logcat の正規表現フラグ -e , --regex= ）。

参考資料

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

2.3.1.3 ProGuard によるログ機能削除

本番リリースの準備として、ProGuard (Android Studio に含まれる) などのツールを使用することができる。android.util.Log クラスのすべてのロギング機能が削除されたかどうかを判断するには、ProGuard 構成ファイル (proguard-rules.pro) で次のオプションを確認する（ロギングコードを削除するこの例と、Android Studio プロジェクトでの ProGuard の有効化に関するこの記事に従うこと)。

```
-assumenosideeffects class android.util.Log
{
  public static boolean isLoggable(java.lang.String, int);
  public static int v(...);
  public static int i(...);
  public static int w(...);
  public static int d(...);
  public static int e(...);
  public static int wtf(...);
}
```

上記の例では、Log クラスのメソッドの呼び出しが削除されることだけが保証されていることに注意する。ログに記録される文字列が動的に生成される場合、その文字列を生成するコードはバイトコードに残る可能性がある。例えば、次のコードは暗黙のうちに StringBuilder を発行してログステートメントを生成する。

Java での例：

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

Kotlin での例：

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

ただし、コンパイルされたバイトコードは、文字列を明示的に生成する次のログステートメントのバイトコードと同等である。

Java での例：

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

Kotlin での例：

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

ProGuard は、Log.v メソッド呼び出しの削除を保証する。残りのコード (new StringBuilder ...) が削除されるかは、コードの複雑さと ProGuard のバージョンに依存する。これは、(未使用の) 文字列が平文データをメモリ上に漏洩させ、デバッガやメモリダンプによってアクセスされる可能性があるため、セキュリティ上のリスクとなる。この問題に対するクリティカルな対処法は存在しないが、1 つの選択肢としては単純な引数を取得し、ログステートメントを内部的に構築するカスタムロギング機能を実装する方法がある。

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

その後、ProGuard がその呼び出しを除去するように設定する。

参考資料

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

2.3.2 ルールブック

1. ログを出力する場合は出力内容に機密情報を含めない（必須）

2.3.2.1 ログを出力する場合は出力内容に機密情報を含めない（必須）

ログ出力をする場合は、出力内容に機密情報が含まれていないことを確認する必要がある。

一般的なログ出力用クラスとしては、以下が存在する。

- Log
- Logger

Log クラス

android.util パッケージに含まれるログ出力用のクラスで、Log.v(), Log.d(), Log.i(), Log.w(), Log.e() メソッドを使用してログを書き込む。書き込んだログは Logcat 上で確認することができる。

各メソッドはログのレベルごとに区分けされている。以下はログのレベルと、それに紐づくメソッドの一覧である。

表 2.3.2.1.1　ログレベルと Log メソッドの一覧

No	ログレベル	メソッド
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

下記は Log クラスによるログ出力コードの一例。

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

Logger クラス

java.util.logging に含まれるログ出力用のクラスで、特定のシステムまたはアプリケーションコンポーネントのメッセージをログに記録するために使用する。通常、階層的なドット区切りの名前空間を使用して名前が付けられる。Logger 名は任意の文字列にすることができるが、通常はログに記録されるコンポーネントのパッケージ名またはクラス名（ java.net や javax.swing など）に基づいている必要がある。

下記は Logger クラスによるログ出力コードの一例。

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

これに違反する場合、以下の可能性がある。

- 第三者に機密情報を読み取られる。

2.4 MSTG-STORAGE-4

機密データはアーキテクチャに必要な部分でない限りサードパーティと共有されていない。

2.4.1 アプリデータの共有

機密情報がサードパーティに漏洩する可能性の一例として、以下のような手段がある。

参考資料

- 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 サードパーティサービスへのデータの共有

サードパーティサービスが提供する機能には、アプリ使用中のユーザの行動を監視するためのトラッキングサービス、バナー広告の販売、またはユーザエクスペリエンスの向上が含まれる。

欠点として、通常開発者は利用するサードパーティライブラリを介して実行されるコードの詳細を把握することができない。したがって、必要以上の情報をサービスに送信したり、機密情報を公開すべきではない。

サードパーティサービスの多くは、以下の 2 つの方法で実装されている。

- スタンドアローンライブラリ
- full SDK を使用

静的解析サードパーティライブラリが提供する API 呼び出しと関数がベストプラクティスに従って使用されているかどうかを判断するには、ソースコードを確認し、アクセス許可を要求し、既知の脆弱性が存在しないかを確認する（「**サードパーティライブラリ使用時の注意点（ *MSTG-CODE-5* ）**」を参照）。

サードパーティサービスに送信される全てのデータは、サードパーティがユーザアカウントを識別できるようにする PII（個人識別情報）の公開を防ぐために、匿名化する必要がある。その他のデータ（ユーザアカウントまたはセッションにマッピングできる ID など）をサードパーティに送信しないようにすること。

動的解析機密情報が埋め込まれていないか、外部サービスへのすべてのリクエストを確認する。クライアントとサーバ間のトラフィックを傍受するには、Burp Suite Professional または OWASP ZAP を使用して中間者（ MITM ）攻撃を開始することにより、動的分析を実行できる。傍受プロキシを介してトラフィックをルーティングすることで、アプリとサーバの間を通過するトラフィックを傍受できる。メイン関数がホストされているサーバに直接送信されない全てのアプリリクエストは、トラッカーや広告サービスの PII などの機密情報についてチェックする必要がある。

参考資料

- 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

ルールブック

- **サードパーティライブラリへ必要のない機密情報を共有しない（必須）**

2.4.1.2 アプリの通知によるデータの共有

通知はプライベートで利用すべきではないことを理解することが重要である。通知が Android システムによって処理されると、システム全体にブロードキャストされ、NotificationListenerService で実行されているアプリケーションはこれらの通知を受信し、必要に応じて処理することができる。

Joker や Alien など、NotificationListenerService を悪用してデバイスの通知を受信し、攻撃者が管理する C2 インフラストラクチャに送信するマルウェアのサンプルが多数存在する。一般的にこれは、デバイス上の通知として表示される二要素認証（ 2FA ）コードを受信し、それを攻撃者に送信する。ユーザにとってより安全な代替手段は、通知を生成しない 2FA アプリケーションを使用することである。

さらに、Google Play ストアには、基本的に Android システム上の全ての通知をローカルに記録する通知ログを提供するアプリが多数存在する。これは、Android では通知が決してプライベートなものではなく、デバイス上の他のアプリからアクセス可能なことを強調している。

そのため、悪意のあるアプリケーションによって使用される可能性のある機密情報やリスクの高い情報がないか、全ての通知の使用を検証する必要がある。

静的解析何らかの通知管理処理に用いられる可能性のある NotificationManager クラスの使用を確認する。このクラスが使用されている場合は、次にアプリケーションがどのように通知を生成し、どのデータが最終的に表示されるかを理解する必要がある。

動的解析アプリケーションを実行し、NotificationCompat.Builder の setContentTitle や setContentText など、通知の作成に関連する関数の全呼び出しをトレースする。その後トレース結果を確認し、他のアプリが盗聴した可能性のあるデータに機密情報が含まれているかどうかを評価する。

参考資料

- 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

ルールブック

- 通知に機密情報を含めない（必須）

2.4.2 ルールブック

1. サードパーティライブラリへ必要のない機密情報を共有しない（必須）
2. 通知に機密情報を含めない（必須）

2.4.2.1 サードパーティライブラリへ必要のない機密情報を共有しない（必須）

サードパーティライブラリを使用する場合は、ライブラリへ渡すパラメータとして、必要でない機密情報が設定されていないことを確認する。必要でない機密情報が設定されている場合、ライブラリ内の処理で悪質に利用される可能性があるため注意する。

通信用のライブラリでは必要とされる場合があるため、上記懸念を考慮する場合は、事前にサーバ・クライアント間で決めた暗号化方式により機密情報を暗号化して、ライブラリに渡す等の対応が必要である。

これに違反する場合、以下の可能性がある。

- サードパーティライブラリの処理で悪用される可能性がある。

2.4.2.2 通知に機密情報を含めない（必須）

発生したイベントをユーザに通知するためには NotificationManager クラスを使用する。

通知の構成要素（表示内容）は NotificationCompat.Builder オブジェクトに指定する。NotificationCompat.Builder クラスには通知の構成要素を指定するためのメソッドが用意されている。下記は指定用メソッドの一例。

- setContentTitle：標準通知で、通知のタイトル (最初の行) を指定する。
- setContentText：標準通知で、通知のテキスト (2 行目) を指定する。

通知を使用する場合は、setContentTitle, setContentText に機密情報が設定されていないことに注意する。

下記は NotificationCompat.Builder クラスへ通知の構成要素を指定し、NotificationManager クラスにより通知を表示するソースコードの一例。

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

これに違反する場合、以下の可能性がある。

- 第三者に機密情報を読み取られる。

2.5 MSTG-STORAGE-5

機密データを処理するテキスト入力では、キーボードキャッシュが無効にされている。

2.5.1 機密データの自動入力

ユーザが入力フィールドに入力すると、ソフトウェアが自動的にデータをサジェストする。この機能は、メッセージアプリにおいて非常に便利である。しかし、ユーザがこのタイプの情報を取得する入力フィールドを選択した場合、キーボードキャッシュが機密情報を開示する可能性がある。

静的解析 Activity のレイアウト定義では、XML 属性を持つ TextView を定義できる。XML 属性 android:inputType に値 textNoSuggestions が指定されている場合、入力フィールドが選択されたときにキーボードキャッシュは表示されなくなる。そのため、ユーザは全てを手動で入力する必要がある。

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

機密情報を入力するすべての入力フィールドのコードには、キーボードによるサジェストを無効にするために、この XML 属性を含める必要がある。

動的解析アプリを起動し、機密データを取得する入力フィールドをクリックする。その際に文字列がサジェストされた場合、これらのフィールドのキーボードキャッシュは無効になっていない。

参考資料

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

ルールブック

- **機密情報を入力するすべての入力フィールドのコードは、キーボードによるサジェストが無効となるように実装する（必須）**
- **機密情報を入力するすべての入力フィールドのレイアウトは、キーボードによるサジェストが無効となるように実装する（必須）**

2.5.2 ルールブック

1. **機密情報を入力するすべての入力フィールドのコードは、キーボードによるサジェストが無効となるように実装する（必須）**
2. **機密情報を入力するすべての入力フィールドのレイアウトは、キーボードによるサジェストが無効となるように実装する（必須）**

2.5.2.1 機密情報を入力するすべての入力フィールドのコードは、キーボードによるサジェストが無効となるように実装する（必須）

テキスト入力および変更をする場合に EditText クラスを使用する。テキスト編集ウィジェットを定義する場合、android.R.styleable#TextView_inputType 属性を設定する必要がある。

機密情報を入力するフィールドのコードでは、inputType 属性へ TYPE_TEXT_FLAG_NO_SUGGESTIONS フラグを設定する。ただし、パスワードや PIN を入力するフィールドの場合には、マスキング用に inputType 属性へ TYPE_TEXT_VARIATION_PASSWORD フラグを設定 (入力テキスト参照) する。

下記はコード上で inputType 属性のフラグとして TYPE_TEXT_FLAG_NO_SUGGESTIONS を設定するコードの一例。

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

また、キャッシュを再度有効にする値で上書きしていないか確認する必要がある。

これに違反する場合、以下の可能性がある。

- 第三者に機密情報を読み取られる。

2.5.2.2 機密情報を入力するすべての入力フィールドのレイアウトは、キーボードによるサジェストが無効となるように実装する（必須）

機密情報を入力するフィールドのレイアウト（ EditText ）では、inputType 属性へ "textNoSuggestions" を設定する。ただし、パスワードや PIN を入力するフィールドの場合には、マスキング用に inputType 属性へ "textPassword" を設定 (入力テキスト参照) する。

下記はコード上で inputType 属性として "textNoSuggestions" を設定するコードの一例。

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

これに違反する場合、以下の可能性がある。

- 第三者に機密情報を読み取られる。

2.6 MSTG-STORAGE-6

機密データは IPC メカニズムを介して公開されていない。

2.6.1 ContentProvider による機密データへのアクセス

Android の IPC メカニズムの一部として、ContentProvider はアプリの保存データに他のアプリがアクセスして変更できるよう許可する。適切に設定されていない場合、これらのメカニズムから機密データが漏洩する可能性がある。

静的解析 AndroidManifest.xml を調べることで、アプリが公開する ContentProvider を検出することができる。ContentProvider は、<provider> 要素で特定できる。

- export タグ (android:exported) の値が「 true 」であるかどうかを判定する。そうでない場合でも、タグに <intent-filter> が定義されていれば、タグは自動的に「 true 」が設定される。アプリからのアクセスのみを想定している場合は、android:exported を「 false 」に設定する。そうでない場合は、フラグを「 true 」に設定し、適切な読み取り/書き込み権限を定義する。
- データが permission タグ (android:permission) によって保護されているかどうかを判断する。permission タグは、他のアプリへの公開を制限する。
- android:protectionLevel 属性の値に signature があるかどうかを判断する。この設定は、同じ企業のアプリのみがデータにアクセスすることを意図していることを示す (つまり、同じキーで signature されている)。他のアプリからデータにアクセスできるようにするには、<permission> 要素でセキュリティポリシーを適用し、適切な android:protectionLevel を設定する。android:permission を使用する場合、他のアプリケーションは ContentProvider にアクセスするために、manifest で対応する <uses-permission> 要素を宣言する必要がある。android:grantUriPermissions 属性を使用して、他のアプリケーションにより具体的なアクセスを許可することができる。<grant-uri-permission> 要素を使用してアクセスを制限することができる。

ソースコードを調べて、ContentProvider の使用方法を把握する。次のキーワードを検索する。

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

※アプリ内での SQL インジェクション攻撃を回避するには、query, update, delete などのパラメータ化されたクエリメソッドを使用する。すべてのメソッド引数を適切にサニタイズすること。例えば、selection 引数が連結されたユーザ入力で構成されている場合、SQL インジェクションにつながる可能性がある。

ContentProvider を公開する場合は、パラメータ化されたクエリメソッド（ query, update, delete ）を使用して

SQL インジェクションを防止しているかどうかを確認する。その場合は、すべての引数が適切にサニタイズされていることを確認すること。

脆弱性のある ContentProvider の例として、脆弱性のあるパスワードマネージャアプリ「Sieve」が存在する。

Android Manifest の検証定義された全ての <provider> 要素を特定する。

```
<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"
/>
```

上記の AndroidManifest.xml に示されているように、アプリケーションは 2 つの ContentProvider をエクスポートする。その内パス（"/Keys"）は、読み取りと書き込みのアクセス許可によって保護されていることに注意すること。

ソースコードの検証 DBContentProvider.java ファイルのクエリ関数を調べて、機密情報が漏洩していないかどうかを確認する。

Java での例：

```
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);
}
```

Kotlin での例：

```
fun query(uri: Uri?, array: Array<String?>?, s: String?, array2: Array<String?>?,␣
↪s2: String?): Cursor {
```

(次のページに続く)

(前のページからの続き)

```
        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)
    }
```

ここでは、実際には "/Keys" と "/Passwords" の 2 つのパスがあり、後者は manifest で保護されていないため、脆弱性があることが確認できる。

URI にアクセスする場合、クエリ文はすべてのパスワードとパス "Passwords/" を返す。これについては「動的解析」セクションで説明し、必要な正確な URI を示す。

動的解析 ContentProvider のテストアプリケーションの ContentProvider を動的に解析するには、まず攻撃対象領域を列挙する。アプリのパッケージ名を Drozer モジュール app.provider.info に渡す。

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

この例では、2 つのコンテンツ プロバイダーがエクスポートされている。DBContentProvider の "/Keys" パスを除いて、どちらも許可なくアクセスができる。この情報を使用して、コンテンツ URI の一部を再構築して DBContentProvider にアクセスできる（ URI は content:// で始まる）。

アプリケーション内で ContentProvider の URI を識別するには、Drozer の scanner.provider.finduris モジュールを使用する。このモジュールは、いくつかの方法でパスを推測し、アクセス可能なコンテンツ URI を決定する。

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

(次のページに続く)

(前のページからの続き)

```
...
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/
```

アクセス可能な ContentProvider のリストを取得したら、app.provider.query モジュールを使用して各プロバイダからデータを抽出してみること。

```
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
```

また Drozer を使用して、脆弱な ContentProvider からレコードを insert, update、および delete することもできる。

- Insert レコード

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

- Update レコード

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

- Delete レコード

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

ContentProvider における SQL インジェクション Android プラットフォームは、ユーザデータを格納するために SQLite データベースを推進している。これらのデータベースは SQL に基づいているため、SQL インジェクションに対して脆弱である可能性がある。Drozer モジュール app.provider.query を使用して、ContentProvider に渡される射影および選択フィールドを操作することにより、SQL インジェクションをテストできる。

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

(次のページに続く)

(前のページからの続き)

```
↪Passwords

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

アプリケーションが SQL インジェクションに対して脆弱な場合、詳細なエラーメッセージが返される。Android の SQL インジェクションは、脆弱な ContentProvider からのデータの変更、またはクエリするために使用される可能性がある。次の例では、Drozer モジュール app.provider.query を使用して、全てのデータベーステーブルを一覧表示する。

```
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 インジェクションを使用して、保護されていないテーブルからデータを取得することもできる。

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

アプリ内の脆弱な ContentProvider を自動的に検出する scanner.provider.injection モジュールを使用して、これらの手順を自動化できる。

```
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/
```

ファイルシステムベースの ContentProvider ContentProvider は、基盤となるファイルシステムへのアクセスを提供できる。これにより、アプリはファイルを共有できる（通常、Android サンドボックスはこれを防ぐことができる）。Drozer モジュール app.provider.read および app.provider.download を使用して、エクスポートされたファイルベースの ContentProvider からファイルをそれぞれ読み取りおよびダウンロードできる。これらの ContentProvider はディレクトリトラバーサルの影響を受けやすく、ターゲットアプリケーションのサンドボックス内で保護されているファイルが読み取られる可能性がある。

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

scanner.provider.traversal モジュールを使用して、ディレクトリトラバーサルの影響を受けやすい ContentProvider を見つけるプロセスを自動化することができる。

```
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
```

adb は、ContentProvider のクエリにも使用できることに注意すること。

```
$ 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
...
```

参考資料

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

ルールブック

- *ContentProvider* のアクセス権限を適切に設定する（必須）
- *SQL* データベース利用時は *SQL* インジェクションを対策する（必須）
- *ContentProvider* 利用時はディレクトリトラバーサルを対策する（必須）

2.6.2 ルールブック

1. *ContentProvider* のアクセス権限を適切に設定する（必須）
2. *SQL* データベース利用時は *SQL* インジェクションを対策する（必須）
3. *ContentProvider* 利用時はディレクトリトラバーサルを対策する（必須）

2.6.2.1 ContentProvider のアクセス権限を適切に設定する（必須）

アプリ内のすべての ContentProvider は、AndroidManifest.xml 内の <provider> 要素で定義する必要がある。未定義の場合、システムは ContentProvider を認識せず、実行しない。

対象アプリの一部である ContentProvider のみを宣言し、対象アプリ内で使用している ContentProvider であっても、他のアプリの一部であるものは宣言しないこと。

下記は AndroidManifest.xml での <provider> 要素の定義の一例。

```
<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>
```

アクセス対象のデータを考慮してコンテンツプロバイダーの設定を適切に行う必要がある。

ContentProvider を他のアプリが使用できるか設定するタグ android:exported このタグでは、他のアプリが ContentProvider を使用できるか設定できる。下記が設定できる設定値である。

- true：他のアプリが ContentProvider を使用できる。どのようなアプリでも、ContentProvider に指定されている権限に従い、ContentProvider のコンテンツ URI を使用して ContentProvider にアクセスできる。
- false：他のアプリが ContentProvider を使用できません。android:exported="false" を設定すると、ContentProvider へのアクセスが対象アプリに限定される。これが設定されている場合に ContentProvider へアクセスできるアプリは、ContentProvider と同じユーザ ID（UID）を持つアプリか、android:grantUriPermissions タグによって一時的にアクセス権を付与されたアプリに限定される。

このタグは API level 17 で導入されたため、API level 16 以下を搭載しているすべてのデバイスは、このタグが "true" に設定されている場合と同じに動作となる。android:targetSdkVersion を 17 以上に設定した場合、API level 17 以上を搭載しているデバイスでは、デフォルト値が "false" となる。

※ exported が false の場合でも、<intent-filter> が定義されている場合は exported に true が設定されている場合と同じ状態となるため、注意する。

ContentProvider のデータ読み書きで必要とする <permission> 要素名を設定するタグ android:permission

ContentProvider のデータを読み書きする際にクライアントが必要とする <permission> 要素名を設定する。この属性は、読み取りと書き込みの両方に対して 1 つの権限を設定する際に便利である。ただし、この属性より

も android:readPermission 属性、android:writePermission 属性、android:grantUriPermissions 属性の方が優先される。android:readPermission 属性も設定した場合、ContentProvider に対してクエリを行うためのアクセスが制御される。また、android:writePermission 属性を設定した場合、ContentProvider のデータを変更するためのアクセスが制御される。

<permission> 要素内の android:protectionLevel タグへ保護レベルを設定することで、権限に含まれている可能性があるリスクと、権限をリクエスト元のアプリに付与するかどうかを決める際にシステムが従う必要がある手順を指定できる。

各保護レベルは、基本権限タイプと protectionLevel を指定する。下記が基本権限タイプの一覧である。

表 2.6.2.1.1　protectionLevel の基本権限タイプ一覧

基本権限タイプ	説明
normal	デフォルト値。分離されたアプリレベルの機能へのアクセスをリクエスト元のアプリに提供する低リスクの権限。
dangerous	リクエスト元のアプリによる個人データへのアクセスあるいはデバイスの管理を許し、ユーザに悪影響を及ぼしかねない高リスクの権限。
signature	権限を宣言したアプリと同じ証明書がリクエスト元のアプリの署名に使用されている場合にのみシステムから付与される権限。
signatureOrSystem	Android システムイメージの専用フォルダにインストールされているアプリ、または権限を宣言したアプリと同じ証明書を使用して署名されたアプリにのみシステムから付与される権限。なお、API level 23 で非推奨になった。

一時的に ContentProvider のデータへのアクセスを許可するタグ android:grantUriPermissions

ContentProvider のデータにアクセスする権限を持たないユーザに対して、そのような権限を付与できるかを設定する。付与した場合、android:readPermission 属性、android:writePermission 属性、android:permission 属性、android:exported 属性によって課される制限が一時的に解除される。権限を付与できる場合は「 true 」、そうでない場合は「 false 」に設定する。「 true 」に設定した場合、ContentProvider の任意のデータに対して権限を付与できる。「 false 」に設定した場合、サブ要素内にリストされるデータサブセット（存在する場合）に対してのみ権限を付与できる。デフォルト値は「false」である。

これに違反する場合、以下の可能性がある。

- 意図せず他のアプリに機密データが漏洩する可能性がある。

2.6.2.2 SQL データベース利用時は SQL インジェクションを対策する（必須）

ContentProvider で SQL データベースを利用する場合は SQL インジェクションを対策する必要がある。

以下へ対策方法を記載する。

1. ContentProvider を他のアプリにエクスポーズする必要がない場合:

    - マニフェスト内で、対象 ContentProvider の <provider> タグを変更して、android:exported="false" に設定する。これにより、他のアプリは対象 ContentProvider にインテントを送信できなくなる。

- android:permission 属性を android:protectionLevel="signature" の permission に設定することで、他のデベロッパーが記述したアプリが対象の ContentProvider にインテントを送信できないようにすることもできる。

2. ContentProvider を他のアプリにエクスポーズする必要がある場合:

   rawQuery() メソッドへ渡す sql を事前にバリデーションチェックし、不要な文字のエスケープを行う。また、置換可能なパラメータとして？を選択句と独立した選択引数の配列で使用すると、ユーザ入力が SQL ステートメントの一部として解釈されるのではなくクエリに直接束縛され、これによりリスクが軽減される。以下へサンプルコードを示す。

```
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;
}
```

これに違反する場合、以下の可能性がある。

- SQL インジェクションの脆弱性を悪用される可能性がある。

2.6.2.3 ContentProvider 利用時はディレクトリトラバーサルを対策する（必須）

ContentProvider を利用する場合はディレクトリトラバーサルを対策する必要がある。

以下へ対策方法を記載する。

1. ContentProvider を他のアプリにエクスポーズする必要がない場合:

   - マニフェスト内で、対象 ContentProvider の <provider> タグを変更して、android:exported="false" に設定する。これにより、他のアプリは対象 ContentProvider にインテントを送信できなくなる。

   - android:permission 属性を android:protectionLevel="signature" の permission に設定することで、他のデベロッパーが記述したアプリが対象の ContentProvider にインテントを送信できないようにすることもできる。

2. ContentProvider を他のアプリにエクスポーズする必要がある場合:

openFile に対する入力がパス トラバーサル文字を含むときに、アプリが絶対に想定外のファイルを返すことのないように、正しく設定する必要がある。そのためには、ファイルの正規パス（ canonical path ）をチェックする。以下へサンプルコードを示す。

```
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);
}
```

これに違反する場合、以下の可能性がある。

- ディレクトリトラバーサルの脆弱性を悪用される可能性がある。

2.7 MSTG-STORAGE-7

パスワードや PIN などの機密データは、ユーザインタフェースを介して公開されていない。

2.7.1 ユーザインターフェースでの機密データの公開

アカウント登録や支払いなど、多くのアプリケーションを利用する際に、機密情報を入力することは不可欠である。このデータは、クレジットカードのデータやユーザアカウントのパスワードなどの金融情報である場合がある。このようなデータは、入力中にアプリが適切にマスクしなければ、漏洩する可能性がある。

情報漏洩を防ぎ、shoulder surfing のようなリスクを軽減するために、明示的に要求されない限り（例：パスワードの入力)、ユーザインターフェースを通じて機密データが公開されないことを確認する必要がある。必要なデータについては、平文の代わりにアスタリスクやドットを表示するなどして、適切にマスキングする必要がある。

そのような情報を表示する、あるいは入力として受け取るすべての UI コンポーネントを注意深く確認すること。機密情報の痕跡を探し、それをマスクするか完全に削除するかを評価する

参考資料

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

2.7.1.1 入力テキスト

静的解析アプリケーションが機密性の高いユーザ入力をマスキングしているかどうかを確認するには、EditText の定義に以下の属性があるかを確認する。

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

この設定により、テキストフィールドに（入力文字ではなく）ドットが表示され、アプリからユーザインターフェースへのパスワードや PIN の漏洩を防ぐことができる。

動的解析入力をアスタリスクやドットに置き換えることで情報がマスクされている場合、アプリはユーザインターフェースにデータを漏洩していないことになる。

参考資料

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

ルールブック

- **機密情報であるパスワードや *PIN* を入力するフィールドではマスキングを行う（必須）**

2.7.1.2 アプリ通知

静的解析アプリケーションを静的に評価する場合、何らかの通知管理の形式である可能性のある NotificationManager クラスの使用を検索することを推奨する。このクラスが使用されている場合、次のステップは、アプリケーションがどのように通知を生成しているかを理解することである。

これらのコード位置は、以下の動的解析セクションに入力することができ、アプリケーションのどこで通知が動的に生成されるかを把握することができる。

動的解析

通知の使い方を特定するために、アプリケーション全体とその利用可能なすべての機能を通じて、通知をトリガーする方法を探す。特定の通知をトリガーするために、アプリケーションの外部でアクションを実行する必要があるかもしれないことを考慮すること。

アプリケーションの実行中に、NotificationCompat.Builder の setContentTitle や setContentText など、通知の作成に関連する関数へのすべての呼び出しをトレースし始めるとよい。最終的にトレースを観察し、機密情報が含まれているかどうかを評価すること。

参考資料

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

ルールブック

- **アプリケーションがどのように通知を生成し、どのデータを表示するか理解した上で実装する（必須）**

2.7.2 ルールブック

1. 機密情報であるパスワードや *PIN* を入力するフィールドではマスキングを行う（必須）
2. アプリケーションがどのように通知を生成し、どのデータを表示するか理解した上で実装する（必須）

2.7.2.1 機密情報であるパスワードや PIN を入力するフィールドではマスキングを行う（必須）

機密情報であるパスワードや PIN は表示されることで漏洩に繋がる。そのため、フィールドではマスキング・非表示にする必要がある。

以下へ入力フィールドをマスキングする方法を示す。

レイアウトの場合：

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

コードの場合：

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

これに違反する場合、以下の可能性がある。

- 第三者に機密情報を読み取られる。

2.7.2.2 アプリケーションがどのように通知を生成し、どのデータを表示するか理解した上で実装する（必須）

発生したイベントをユーザに通知するためには NotificationManager クラスを使用する。

通知の構成要素（表示内容）は NotificationCompat.Builder オブジェクトに指定する。NotificationCompat.Builder クラスには通知の構成要素を指定するためのメソッドが用意されている。下記は指定用メソッドの一例。

- setContentTitle：標準通知で、通知のタイトル (最初の行) を指定する。
- setContentText：標準通知で、通知のテキスト (2 行目) を指定する。

下記は NotificationCompat.Builder クラスへ通知の構成要素を指定し、NotificationManager クラスにより通知を表示するソースコードの一例。

```
    var builder = NotificationCompat.Builder(this, CHANNEL_ID)
            .setSmallIcon(R.drawable.notification_icon)
            .setContentTitle(textTitle)
            .setContentText(textContent)
            .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    with(NotificationManagerCompat.from(this)) {
```

(次のページに続く)

(前のページからの続き)

```
        // notificationID と builder.build() を渡します
        notify(notificationID, builder.build())
    }
```

これに違反する場合、以下の可能性がある。

- 第三者に機密情報を読み取られる。

2.8 MSTG-STORAGE-12

アプリは処理される個人識別情報の種類をユーザに通知しており、同様にユーザがアプリを使用する際に従うべきセキュリティのベストプラクティスについて通知している。

2.8.1 アプリマーケットプレイスでのデータプライバシーに関するユーザ教育のテスト

現時点では、どのプライバシー関連情報が開発者によって開示されているかを知り、それが妥当であるかどうかを評価しようとしているだけである (アクセス許可をテストするときと同様)。

実際に収集または共有されている特定の情報を開発者が宣言していない可能性があるが、それはここでこのテストを拡張する別のトピックのためのものである。このテストの一環として、プライバシー違反の保証を提供することは想定されていない。

参考資料

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

2.8.2 静的解析

以下の手順で実行できる。

1. 対応するアプリマーケットプレイス（ Google Play, App Store など）でアプリを検索する。
2. "Privacy Details" セクション（ App Store ）または "Safety Section" セクション（ Google Play ）に移動する。
3. 利用可能な情報があるかどうかを確認する。

開発者がアプリマーケットプレイスのガイドラインに従ってコンパイルし、必要なラベルと説明を含めた場合、テストは合格である。アプリマーケットプレイスから取得した情報を証拠として保存・提供し、後でそれを使用してプライバシーまたはデータ保護の潜在的な違反を評価できるようにする。

参考資料

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

2.8.3 動的解析

オプションの手順として、このテストの一部として何らかの証拠を提供することもできる。例えば、iOS アプリをテストしている場合、アプリのアクティビティの記録を有効にして、写真、連絡先、カメラ、マイク、ネットワーク接続などのさまざまなリソースへの詳細なアプリアクセスを含むプライバシーレポートを簡単にエクスポートできる。

これを行うと、実際には他の MASVS カテゴリをテストする際に多くの利点がある。これは、MASVS-NETWORK でのネットワーク通信のテストや、MASVS-PLATFORM でのアプリのパーミッションのテストに使用できる非常に役立つ情報を提供する。これらの他のカテゴリをテストしているときに、他のテストツールを使用して同様の測定を行った可能性がある。これをこのテストの証拠として提供することもできる。

理想的には、利用可能な情報を、アプリが実際に意図していることと比較する必要がある。ただし、リソースや自動化されたツールのサポートによっては、完了するまでに数日から数週間かかる可能性がある簡単なタスクではない。また、アプリの機能とコンテキストに大きく依存するため、理想的には、アプリ開発者と密接に連携するホワイトボックスセットアップで実行する必要がある。

参考資料

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

2.8.4 セキュリティのベストプラクティスに関するユーザ教育のテスト

このテストは、自動化を意図している場合は特に難しいかもしれない。アプリを広く使い、以下の質問に答えられるようにすることを推奨する。

- 指紋の使用：高リスクの取引／情報へのアクセスを提供する認証に指紋が使用される。アプリケーションは、デバイスに他の人の指紋が複数登録されている場合に起こりうる問題について、ユーザに通知しているか？
- Root 化 /Jailbreak：root 化または Jailbreak 検出が実装されている。アプリケーションは、特定のリスクの高いアクションがデバイスの Jailbreak/root 化ステータスのために追加のリスクを伴うという事実をユーザに通知しているか？
- 特定の認証情報：ユーザがアプリケーションからリカバリーコード、パスワード、PIN を取得する（または設定）。アプリケーションからリカバリーコード、パスワード、PIN を取得（または設定）した場合、アプリケーションはこれを他のユーザと決して共有せず、アプリケーションのみが要求するようユーザに指示しているか？
- アプリケーションの配布：リスクの高いアプリケーションの場合、ユーザが危険なバージョンのアプリケーションをダウンロードするのを防ぐため。アプリケーションの製造元は、アプリケーション配布する正式な方法（ Google Play ）を適切に伝えているか？
- Prominent Disclosure：全てのケース。アプリケーションは、データへのアクセス、収集、使用、共有について、目立つように開示しているか？ 例えば、アプリケーションは、Android 上で許可を求めるために Best practices for prominent disclosure and consent を使用しているか。

参考資料

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

ルールブック

- アプリを広く使い、セキュリティのベストプラクティスに関する質問に答えられるようにする（推奨）

2.8.5 ルールブック

1. アプリを広く使い、セキュリティのベストプラクティスに関する質問に答えられるようにする（推奨）

2.8.5.1 アプリを広く使い、セキュリティのベストプラクティスに関する質問に答えられるようにする（推奨）

アプリを広く使い、セキュリティのベストプラクティスに関する以下の質問に答えられるようにすることを推奨する。

- 指紋の使用：高リスクの取引／情報へのアクセスを提供する認証に指紋が使用される。アプリケーションは、デバイスに他の人の指紋が複数登録されている場合に起こりうる問題について、ユーザに通知しているか？
- Root 化 /Jailbreak：root 化または Jailbreak 検出が実装されている。アプリケーションは、特定のリスクの高いアクションがデバイスの Jailbreak/root 化ステータスのために追加のリスクを伴うという事実をユーザに通知しているか？
- 特定の認証情報：ユーザがアプリケーションからリカバリーコード、パスワード、PIN を取得する（または設定）。アプリケーションからリカバリーコード、パスワード、PIN を取得（または設定）した場合、アプリケーションはこれを他のユーザと決して共有せず、アプリケーションのみが要求するよう指示しているか？
- アプリケーションの配布：リスクの高いアプリケーションの場合、ユーザが危険なバージョンのアプリケーションをダウンロードするのを防ぐため。アプリケーションの製造元は、アプリケーション配布する正式な方法（ Google Play ）を適切に伝えているか？
- Prominent Disclosure：全てのケース。アプリケーションは、データへのアクセス、収集、使用、共有について、目立つように開示しているか？ 例えば、アプリケーションは、Android 上で許可を求めるために Best practices for prominent disclosure and consent を使用しているか。

これに注意しない場合、以下の可能性がある。

- 機密情報が想定していない処理で利用される。
- 第三者に機密情報を読み取られる。

3 暗号化要件

3.1 MSTG-CRYPTO-1

アプリは暗号化の唯一の方法としてハードコードされた鍵による対称暗号化に依存していない。

3.1.1 問題のある暗号化構成

3.1.1.1 不十分なキーの長さ

最も安全な暗号化アルゴリズムであっても、不十分なキーサイズを使用すると、ブルートフォースアタックに対して脆弱になる。

キーの長さが業界標準を満たしていることを確認する。なお日本国内においては「電子政府推奨暗号リスト」掲載の暗号仕様書一覧を確認する。

参考資料

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

ルールブック

- **業界標準を満たしたキーの長さを設定する（必須）**

3.1.1.2 ハードコードされた暗号化鍵による対称暗号化

対称暗号化とキー付きハッシュ (MAC) のセキュリティは、キーの機密性に依存する。キーが公開されると、暗号化によって得られたセキュリティが失われる。これを防ぐには、作成に関与した暗号化データと同じ場所に秘密鍵を保存しないことである。よくある間違いは、静的なハードコードされた暗号化鍵を使用してローカルに保存されたデータを暗号化し、そのキーをアプリにコンパイルすることである。この場合、逆アセンブラを使用できる人であれば誰でもそのキーにアクセスできるようになる。

ハードコードされた暗号化鍵とは、次のことを意味する。

- アプリケーションリソースの一部であること
- 既知の値から導出可能な値であること
- コードにハードコードされていること

まず、ソースコード内にキーやパスワードが保存されていないことを確認する。つまり、ネイティブコード、JavaScript/Dart コード、Java/Kotlin コードをチェックする必要がある。難読化は動的インストルメンテーションによって容易にバイパスされるため、ハードコードされたキーはソースコードが難読化されていても問題があることに注意する。

アプリが双方向 TLS (サーバとクライアントの両方の証明書が検証される) を使用している場合、以下を確認する。

- クライアント証明書のパスワードがローカルに保存されていない、またはデバイスの Keychain にロックされていること。
- クライアント証明書は、すべてのインストール間で共有されていないこと。

アプリが、アプリのデータ内に保存され暗号化されたコンテナに依存する場合、暗号化鍵がどのように使用されるかを確認する。キーラップ方式を使用している場合、各ユーザのマスターシークレットが初期化されていること、またはコンテナが新しいキーで再暗号化されていることを確認する。マスターシークレットや以前のパスワードを使用してコンテナを復号できる場合、パスワードの変更がどのように処理されるかを確認する。

モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイスストレージに保存する必要がある。プラットフォーム固有の API の詳細については、「Android のデータストレージ」の章を参照する。

参考資料

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

ルールブック

- **ソースコード内にキーやパスワードを保存しない（必須）**
- **クライアント証明書のパスワードをローカルに保存しない、またはデバイスの *Keychain* にロックする（必須）**
- **クライアント証明書はすべてのインストール間で共有しない（必須）**
- **コンテナに依存する場合、暗号化鍵がどのように使用されるかを確認する（必須）**
- **モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイスストレージに保存する（必須）**

3.1.1.3 弱いキー生成関数

暗号化アルゴリズム (対称暗号化や一部の MAC など) は、特定のサイズの秘密の入力を想定している。例えば、AES はちょうど 16 バイトのキーを使用する。ネイティブな実装では、ユーザが提供したパスワードを直接入力キーとして使用することがある。ユーザが提供したパスワードを入力キーとして使用する場合、以下のような問題がある。

- パスワードがキーよりも小さい場合、完全なキースペースは使用されない。残りのスペースはパディングされる (パディングのためにスペースが使われることもある)。
- ユーザ提供のパスワードは、現実的には、ほとんどが表示・発音可能な文字で構成される。したがって、256 文字ある ASCII 文字の一部だけが使われ、エントロピーはおよそ 4 分の 1 に減少する。

パスワードが暗号化関数に直接渡されないようにする。代わりに、ユーザが提供したパスワードは、暗号化鍵を作成するために KDF に渡されるべきである。パスワード導出関数を使用する場合は、適切な反復回数を選択する。例えば、NIST は PBKDF2 の反復回数を少なくとも 10,000 回、ユーザが感じるパフォーマンスが重要でない重要なキーの場合は少なくとも 10,000,000 回 を推奨している。重要なキーについては、Argon2 のような Password Hashing Competition (PHC) で認められたアルゴリズムの実装を検討することが推奨される。

参考資料

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

ルールブック

- **暗号化アルゴリズム *(対称暗号化や一部の MAC など)* を使用する場合、想定されている特定のサイズの秘密の入力を使用する（必須）**
- **ユーザが提供したパスワードは、暗号鍵を作成するために *KDF* に渡す（必須）**
- **パスワード導出関数を使用する場合は、適切な反復回数を選択する（必須）**

3.1.1.4 弱い乱数ジェネレーター

決定論的なデバイスで真の乱数を生成することは基本的に不可能である。擬似乱数ジェネレーター (RNG) は、擬似乱数のストリーム (あたかもランダムに発生したかのように見える数値のストリーム) を生成することでこれを補う。生成される数値の品質は、使用するアルゴリズムの種類によって異なる。暗号化的に安全な RNG は、統計的ランダム性テストに合格した乱数を生成し、予測攻撃に対して耐性がある。(例: 次に生成される数を予測することは統計的に不可能である)

Mobile SDK は、十分な人工的ランダム性を持つ数値を生成する RNG アルゴリズムの標準的な実装を提供している。利用可能な API については、Android 固有のセクションで紹介する。

参考資料

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

ルールブック

- 十分な人工的ランダム性を持つ数値を生成する *RNG* アルゴリズムの標準的な実装を確認する（必須）

3.1.1.5 暗号化のカスタム実装

独自の暗号関数を開発することは、時間がかかり、困難であり、失敗する可能性が高い。その代わりに、安全性が高いと広く認められている、よく知られたアルゴリズムを使用することができる。モバイル OS は、これらのアルゴリズムを実装した標準的な暗号 API を提供している。

ソースコード内で使用されているすべての暗号化方式、特に機密データに直接適用されている暗号化方式を注意深く検査する。すべての暗号化操作では、Android 標準の暗号化 API を使用する必要がある (これらについては、プラットフォーム固有の章で詳しく説明する)。既知のプロバイダが提供する標準的なルーチンを呼び出さない暗号化操作は、厳密に検査する必要がある。標準的なアルゴリズムが変更されている場合は、細心の注意を払う必要がある。エンコーディングは暗号化と同じではないことに注意する。XOR (排他的論理和) のようなビット操作の演算子を見つけたら、必ずさらに調査する。

暗号化のすべての実装で、以下のことが常に行われていることを確認する必要がある。

- ワーカーキー (AES/DES/Rijndael における中間鍵/派生鍵のようなもの) は、消費後またはエラー発生時にメモリから適切に削除される。
- 暗号の内部状態は、できるだけ早くメモリから削除する。

参考資料

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

ルールブック

- 暗号化に関するすべての実装では適切にメモリ状態を管理する（必須）
- *OS* が提供する業界標準の暗号 *API* を使用する（必須）

3.1.1.6 不適切な AES 構成

Advanced Encryption Standard (AES) は、モバイルアプリにおける対称暗号化の標準として広く受け入れられている。AES は、一連の連鎖的な数学演算に基づく反復型ブロック暗号である。AES は入力に対して可変回数のラウンドを実行し、各ラウンドでは入力ブロック内のバイトの置換と並べ替えが行われる。各ラウンドでは、オリジナルの AES キーから派生した 128 ビットのラウンドキーが使用される。

この記事の執筆時点では、AES に対する効率的な暗号解読攻撃は発見されていない。しかし、実装の詳細やブロック暗号モードなどの設定可能なパラメータには、エラーの余地がある。

弱いブロック暗号モード

ブロックベースの暗号化は、個々の入力ブロック (例: AES は 128 ビットブロック) に対して実行される。平文がブロックサイズより大きい場合、平文は内部で指定された入力サイズのブロックに分割され、各ブロックに対して暗号化が実行される。ブロック暗号操作モード (またはブロックモード) は、前のブロックの暗号化の結果が後続のブロックに影響を与えるかどうかを決定する。

ECB (Electronic Codebook) は、入力を一定サイズのブロックに分割し、同じキーを用いて個別に暗号化する。分割された複数のブロックに同じ平文が含まれている場合、それらは同一の暗号文ブロックに暗号化されるため、データのパターンを容易に特定することができる。また、状況によっては、攻撃者が暗号化されたデータを再生することも可能である。

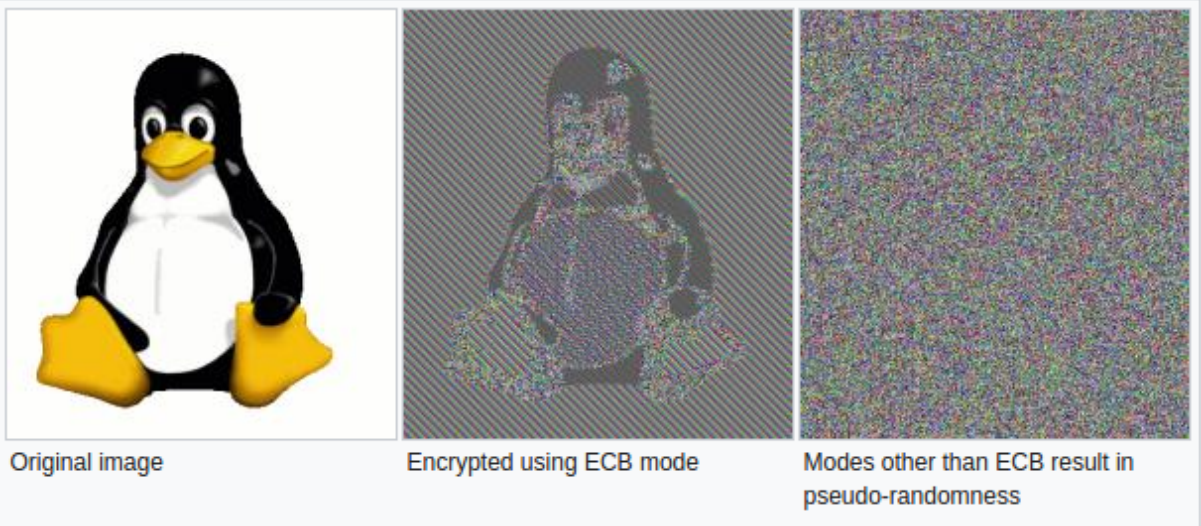


図 3.1.1.6.1　ECB の暗号化例

ECB の代わりに暗号ブロック連鎖 (CBC) モードが使用されていることを確認する。CBC モードでは、平文ブロックは直前の暗号文ブロックと XOR される。これにより、ブロックに同じ情報が含まれている場合でも、暗号化された各ブロックは一意であり、ランダムであることが保証される。CBC を HMAC と組み合わせたり、「パディングエラー」「 MAC エラー」「復号失敗」などのエラーが出ないようにすることが、パディングオラクル攻撃に対抗するために最善であることに注意してください。

暗号化されたデータを保存する場合、Galois/Counter Mode (GCM) のような、保存データの完全性も保護するブロックモードを使用することを推奨する。後者には、このアルゴリズムが各 TLSv1.2 の実装に必須であり、したがってすべての最新のプラットフォームで利用できるという利点もある。

有効なブロックモードの詳細については、ブロックモード選択に関する NIST のガイドラインを参照する。

予測可能な初期化ベクトル

CBC 、OFB 、CFB 、PCBC 、GCM モードでは、暗号への初期入力として、初期化ベクトル (IV) が必要である。IV は秘密にする必要はないが、予測可能であってはならない。暗号化されたメッセージごとにランダムで一意であり、非再現性である必要がある。IV は暗号的に安全な乱数ジェネレーターを用いて生成されていることを確認する。IV の詳細については、Crypto Fail の初期化ベクトルに関する記事を参照する。

コードで使用されている暗号化ライブラリに注意すること: 多くのオープンソースライブラリは、悪い習慣 (ハードコードされた IV の使用など) に従う可能性のあるドキュメントが提供されている。よくある間違いは、IV 値を変更せずにサンプルコードをコピーアンドペーストすることである。

ステートフル操作モードでの初期化ベクトル

初期化ベクトルがカウンター (CTR と nonce の組み合わせ) であることが多い CTR モードと GCM モード を使用する場合は、IV の使用方法が異なることに注意する。したがって、ここでは、独自のステートフルモデル

を持つ予測可能な IV を使用することが、まさに必要である。CTR では、新しいブロック操作のたびに、新しい nonce とカウンターを入力として使用する。例: 5120 ビット長の平文の場合では、20 個のブロックがあるため、nonce とカウンターで構成される 20 個の入力ベクトルが必要である。一方 GCM では、暗号化操作ごとに IV を 1 つだけ持ち、同じキーで繰り返さないようにする。IV の詳細と推奨事項については、GCM に関する NIST の資料の 8 項を参照する。

参考資料

- 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

ルールブック

- **パディングオラクル攻撃に対抗するために、*CBC* を *HMAC* と組み合わせたりパディングエラー, *MAC* エラー, 復号失敗などのエラーが出ないようにする（必須）**
- **暗号化されたデータを保存する場合、*Galois/Counter Mode (GCM)* のような、保存データの完全性も保護するブロックモードを使用する（推奨）**
- ***IV* は暗号的に安全な乱数ジェネレーターを用いて生成する（必須）**
- **初期化ベクトルがカウンターであることが多い *CTR* モードと *GCM* モード を使用する場合は、*IV* の使用方法が異なることに注意する（必須）**

3.1.1.7 弱いパディングまたはブロック操作の実装によるパディングオラクル攻撃

以前は、非対称暗号を行う際のパディングメカニズムとして、PKCS1.5 パディング (コード: PKCS1Padding) が使われていた。このメカニズムは、パディングオラクル攻撃に対して脆弱である。そのため、PKCS#1 v2.0 (コード: OAEPwithSHA-256andMGF1Padding 、OAEPwithSHA-224andMGF1Padding 、OAEPwithSHA-384andMGF1Padding 、OAEPwithSHA-512andMGF1Padding) に取り込まれた OAEP (Optimal Asymmetric Encryption PaddingOAEPPadding) を使用するのが最適である。なお、OAEP を使用した場合でも、Kudelskisecurity のブログで紹介されている Mangers 攻撃としてよく知られている問題に遭遇する可能性があることに注意する。

注: PKCS #5 を使用する AES-CBC は、実装が「パディングエラー」、「 MAC エラー」、または「復号に失敗しました」などの警告を表示するため、パディングオラクル攻撃に対しても脆弱であることが示されている。例として、The Padding Oracle Attack および The CBC Padding Oracle Problem を参照する。次に、平文を暗号化した後に HMAC を追加することが最善である。結局、MAC に失敗した暗号文は復号する必要がなく、破棄することができる。

参考資料

- 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

ルールブック

- **非対称暗号を行う際のパディングメカニズムとして *PKCS#1 v2.0* に取り込まれた *OAEP* を使用する（必須）**

3.1.1.8 ストレージおよびメモリ内のキーの扱い

メモリダンプが脅威モデルの一部である場合、キーがアクティブに使用された瞬間にキーにアクセスできる。メモリダンプには、ルートアクセス (ルート化されたデバイスやジェイルブレイクされたデバイスなど) が必要であるか、Frida でパッチされたアプリケーション (Fridump などのツールを使用できるように) が必要である。したがって、デバイスでキーがまだ必要な場合は、次のことを考慮するのが最善である。

- リモートサーバのキー: Amazon KMS や Azure Key Vault などのリモート Key Vault を使用することができる。一部のユースケースでは、アプリとリモートリソースの間にオーケストレーションレイヤーを開発することが適切なオプションとなる場合がある。例えば、Function as a Service (FaaS) システム (AWS Lambda や Google Cloud Functions など) 上で動作するサーバレス関数が、API キーやシークレットを取得するためのリクエストを転送するような場合である。その他の選択肢として、Amazon Cognito 、Google Identity Platform 、Azure Active Directory なども存在する。
- ハードウェアで保護された安全なストレージ内のキー: すべての暗号化アクションとキー自体が信頼できる実行環境 (例: Android Keystore を使用) にあることを確認する。詳細については、Android Data Storage の章を参照する。
- エンベロープ暗号化によって保護されたキー: キーが TEE/SE の外部に保存されている場合は、multi-layered 暗号化の使用を検討する。エンベロープ暗号化アプローチ (OWASP Cryptographic Storage Cheat Sheet、Google Cloud Key management guide、AWS Well-Architected Framework guide 参照)、またはデータ暗号化鍵をキー暗号化する HPKE アプローチを使用する。
- メモリ内のキー: キーができるだけ短時間しかメモリに残さないようにし、暗号化操作に成功した後やエラー時にキーをゼロにし、無効化することを考慮する。一般的な暗号化のガイドラインについては、機密データのメモリの消去を参照する。より詳細な情報については、「機密データのメモリのテスト」を参照する。

注: メモリダンプが容易になるため、署名の検証や暗号化に使用される公開鍵以外は、アカウントやデバイス間で同じキーを共有しない。

参考資料

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

ルールブック

- **メモリダンプを考慮してキーを使用する（必須）**

- アカウントやデバイス間で同じキーを共有しない（必須）

3.1.1.9 転送時のキーの扱い

キーをデバイス間で、またはアプリからバックエンドに転送する必要がある場合、トランスポート対称鍵や他のメカニズムによって、適切なキー保護が行われていることを確認する。多くの場合、キーは難読化された状態で共有されるため、簡単に元に戻すことができる。代わりに、非対称暗号化またはラッピングキーが使用されていることを確認する。例えば、対称鍵は非対称鍵の公開鍵で暗号化することができる。

参考資料

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

ルールブック

- トランスポート対称鍵や他のメカニズムによって、適切なキー保護を行う（必須）

3.1.2 ハードコードされた対称暗号化

このテストケースでは、暗号化の唯一の方法として、ハードコードされた対称暗号化に焦点を当てている。以下のチェックを行う必要がある。

- 対称暗号化のすべてのインスタンスを識別する
- 識別されたインスタンスごとに、ハードコードされた対称鍵があるかどうかを確認する。
- ハードコードされた対称暗号化が暗号化の唯一の方法として使用されていないかどうかを確認する。

参考資料

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

ルールブック

- ハードコードされた対称暗号化を暗号化の唯一の方法として使用しない（必須）

3.1.2.1 静的解析

コード内の対称鍵暗号化のすべてのインスタンスを識別し、対称鍵をロードまたは提供するメカニズムを探す。

- 対称アルゴリズム (DES 、AES など)。
- キージェネレーターの仕様 (KeyGenParameterSpec 、KeyPairGeneratorSpec 、KeyPairGenerator 、KeyGenerator 、KeyProperties など)
- java.security.* 、javax.crypto.* 、android.security.* 、android.security.keystore.* をインポートしているクラス。

識別されたインスタンスごとに、使用されている対称鍵が以下であるかどうか確認する。

- アプリケーションリソースの一部でないか
- 既知の値から導き出すことができないか
- コードにハードコードされていないか

ハードコードされた各対称鍵について、セキュリティ上重要なコンテキストで、唯一の暗号化方法として使用されていないことを確認する。

例として、ハードコードされた暗号化鍵の使用状況を確認する方法を示す。最初にアプリを逆アセンブルおよび逆コンパイルして、jadx などを使用して Java コードを取得する。

SecretKeySpec クラスが使われているファイルを、再帰的に grep するか、jadx の検索機能を使って検索する。

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

これにより、SecretKeySpec クラスを使用しているすべてのクラスが返される。これらのファイルを調べて、キー マテリアルを渡すためにどの変数が使用されているかを追跡する。下図は、実稼働中のアプリケーションでこの評価を行った結果である。ハードコードされ、静的バイト配列 Encrypt.keyBytes で初期化された静的暗号化鍵の使用を明確に特定できる。

```
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
```

図 3.1.2.1.1　ハードコードされた暗号化鍵の使用の特定

参考資料

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

3.1.2.2 動的解析

暗号化メソッドのメソッドトレースにより、使用されているキーなどの入出力値を把握することができる。暗号化操作の実行中にファイルシステムアクセスを監視して、キーマテリアルの書き込み先または読み取り元を評価する。例えば、RMS - Runtime Mobile Security の API モニターを使用して、ファイルシステムを監視することができる。

参考資料

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

ルールブック

- 暗号化メソッドのメソッドトレースを実施し、キーマテリアルの書き込み先または読み取り元を確認する（必須）

3.1.3 ルールブック

1. 業界標準を満たしたキーの長さを設定する（必須）
2. ソースコード内にキーやパスワードを保存しない（必須）
3. クライアント証明書のパスワードをローカルに保存しない、またはデバイスの *Keychain* にロックする（必須）
4. クライアント証明書はすべてのインストール間で共有しない（必須）
5. コンテナに依存する場合、暗号化鍵がどのように使用されるかを確認する（必須）
6. モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイスストレージに保存する（必須）
7. 暗号化アルゴリズム *(*対称暗号化や一部の *MAC* など*)* を使用する場合、想定されている特定のサイズの秘密の入力を使用する（必須）
8. ユーザが提供したパスワードは、暗号鍵を作成するために *KDF* に渡す（必須）
9. パスワード導出関数を使用する場合は、適切な反復回数を選択する（必須）
10. 十分な人工的ランダム性を持つ数値を生成する *RNG* アルゴリズムの標準的な実装を確認する（必須）
11. 暗号化に関するすべての実装では適切にメモリ状態を管理する（必須）
12. *OS* が提供する業界標準の暗号 *API* を使用する（必須）
13. パディングオラクル攻撃に対抗するために、*CBC* を *HMAC* と組み合わせたりパディングエラー, *MAC* エラー, 復号失敗などのエラーが出ないようにする（必須）
14. 暗号化されたデータを保存する場合、*Galois/Counter Mode (GCM)* のような、保存データの完全性も保護するブロックモードを使用する（推奨）
15. *IV* は暗号的に安全な乱数ジェネレーターを用いて生成する（必須）

16. 初期化ベクトルがカウンターであることが多い *CTR* モードと *GCM* モード を使用する場合は、*IV* の使用方法が異なることに注意する（必須）
17. 非対称暗号を行う際のパディングメカニズムとして *PKCS#1 v2.0* に取り込まれた *OAEP* を使用する（必須）
18. メモリダンプを考慮してキーを使用する（必須）
19. アカウントやデバイス間で同じキーを共有しない（必須）
20. トランスポート対称鍵や他のメカニズムによって、適切なキー保護を行う（必須）
21. ハードコードされた対称暗号化を暗号化の唯一の方法として使用しない（必須）
22. 暗号化メソッドのメソッドトレースを実施し、キーマテリアルの書き込み先または読み取り元を確認する（必須）

3.1.3.1 業界標準を満たしたキーの長さを設定する（必須）

キーの長さが業界標準を満たしていることを確認する。なお日本国内においては「電子政府推奨暗号リスト」掲載の暗号仕様書一覧を確認する。最も安全な暗号化アルゴリズムであっても、不十分なキーサイズを使用すると、ブルートフォースアタックに対して脆弱になる。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- ブルートフォースアタックに対して脆弱になる。

3.1.3.2 ソースコード内にキーやパスワードを保存しない（必須）

難読化は動的インストルメンテーションによって容易にバイパスされるため、ハードコードされたキーはソースコードが難読化されていても問題がある。そのため、ソースコード（ネイティブコード、JavaScript/Dart コード、Java/Kotlin コード）内にキーやパスワードを保存しない。

※非推奨なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- キーやパスワードが漏洩する。

3.1.3.3 クライアント証明書のパスワードをローカルに保存しない、またはデバイスの Keychain にロックする（必須）

アプリが双方向 TLS (サーバとクライアントの両方の証明書が検証される) を使用している場合、クライアント証明書のパスワードをローカルに保存しない。またはデバイスの Keychain にロックする。

サンプルコードは以下ルールブックを参照。

ルールブック

- *Keychain* に初めてインポートする場合、ユーザへ証明書ストレージを保護するためにロック画面の *PIN* またはパスワードを設定するよう促す（必須）
- *Android* のネイティブなメカニズムが機密情報を特定するかどうかを判断する（必須）

これに違反する場合、以下の可能性がある。

- パスワードが第三者に読み取られ悪用される。

3.1.3.4 クライアント証明書はすべてのインストール間で共有しない（必須）

アプリが双方向 TLS (サーバとクライアントの両方の証明書が検証される) を使用している場合、クライアント証明書はすべてのインストール間で共有しない。

※非推奨なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- パスワードが他アプリに読み取られ悪用される。

3.1.3.5 コンテナに依存する場合、暗号化鍵がどのように使用されるかを確認する（必須）

アプリが、アプリのデータ内に保存された暗号化されたコンテナに依存する場合、暗号化鍵がどのように使用されるかを確認する。

キーラップ方式を使用している場合

以下について確認する。

- 各ユーザのマスターシークレットが初期化されていること
- コンテナが新しいキーで再暗号化されていること

マスターシークレットや以前のパスワードを使用してコンテナを復号できる場合

パスワードの変更がどのように処理されるかを確認する。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- パスワードやマスターシークレットが意図した目的以外で使用される。

3.1.3.6 モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイスストレージに保存する（必須）

モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイスストレージに保存する必要がある。Android プラットフォームでの秘密鍵の保存方法については、「暗号化キーの保存」を参照。

ルールブック

- 暗号化キーの保存方法（必須）

これに違反する場合、以下の可能性がある。

- 秘密鍵が他アプリや第三者に読み取られる。

3.1.3.7 暗号化アルゴリズム (対称暗号化や一部の MAC など) を使用する場合、想定されている特定のサイズの秘密の入力を使用する（必須）

暗号化アルゴリズム (対称暗号化や一部の MAC など) を使用する場合、想定されている特定のサイズの秘密の入力を使用する必要がある。例えば、AES はちょうど 16 バイトのキーを使用する。

ネイティブな実装では、ユーザが提供したパスワードを直接入力キーとして使用することがある。ユーザが提供したパスワードを入力キーとして使用する場合、以下のような問題がある。

- パスワードがキーよりも小さい場合、完全なキースペースは使用されない。残りのスペースはパディングされる (パディングのためにスペースが使われることもある)。
- ユーザ提供のパスワードは、現実的には、ほとんどが表示・発音可能な文字で構成される。したがって、256 文字ある ASCII 文字の一部だけが使われ、エントロピーはおよそ 4 分の 1 に減少する。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- 脆弱なキーが生成される。

3.1.3.8 ユーザが提供したパスワードは、暗号鍵を作成するために KDF に渡す（必須）

暗号化関数を使用する場合、ユーザが提供したパスワードは、暗号鍵を作成するために KDF に渡されるべきである。パスワードが暗号化関数に直接渡されないようにする。代わりに、ユーザが提供したパスワードは、暗号化鍵を作成するために KDF に渡されるべきである。パスワード導出関数を使用する場合は、適切な反復回数を選択する。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- パスワードがキーよりも小さい場合、完全なキースペースは使用されない。残りのスペースはパディングされる。
- エントロピーがおよそ 4 分の 1 に減少する。

3.1.3.9 パスワード導出関数を使用する場合は、適切な反復回数を選択する（必須）

パスワード導出関数を使用する場合は、適切な反復回数を選択する必要がある。例えば、NIST は PBKDF2 の反復回数を少なくとも 10,000 回、ユーザが感じるパフォーマンスが重要でない重要なキーの場合は少なくとも 10,000,000 回 を推奨している。重要なキーについては、Argon2 のような Password Hashing Competition (PHC) で認められたアルゴリズムの実装を検討することが推奨される。

※サーバ側のルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- 脆弱なキーが生成される。

3.1.3.10 十分な人工的ランダム性を持つ数値を生成する RNG アルゴリズムの標準的な実装を確認する（必須）

暗号化的に安全な RNG は、統計的ランダム性テストに合格した乱数を生成し、予測攻撃に対して耐性がある。安全な水準に満たない RNG アルゴリズムにより生成された乱数を使用すると、予測攻撃が成功する可能性が高まる。そのため、十分な人工的ランダム性を持つ数値を生成する RNG アルゴリズムを使用する必要がある。

Android 標準で安全性の高い乱数を生成する API については以下を参照する。

ルールブック

- セキュアな乱数ジェネレーターと設定を使用する（必須）

これに違反する場合、以下の可能性がある。

- 予測攻撃が成功する可能性が高まる。

3.1.3.11 暗号化に関するすべての実装では適切にメモリ状態を管理する（必須）

暗号化に関するすべての実装では、ワーカーキー (AES/DES/Rijndael における中間鍵/派生鍵のようなもの) が消費後またはエラー発生時にメモリから適切に削除する必要がある。また暗号の内部状態も、できるだけ早くメモリから削除する必要がある。

AES 実装と実行後の解放:

```
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
```

(次のページに続く)

(前のページからの続き)

```
    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)

        // 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()
            }
```

(次のページに続く)

(前のページからの続き)

```
            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())

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

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

ルールブック

- **キーマテリアルは、不要になったらすぐにメモリから消去する必要がある（必須）**

これに違反する場合、以下の可能性がある。

- メモリに残された暗号化情報を意図しない処理で利用される。

3.1.3.12 OS が提供する業界標準の暗号 API を使用する（必須）

独自の暗号関数を開発することは、時間がかかり、困難であり、失敗する可能性が高い。その代わりに、安全性が高いと広く認められている、よく知られたアルゴリズムを使用することができる。モバイル OS は、これらのアルゴリズムを実装した標準的な暗号 API を提供しているため、安全な暗号化のためにはこちらを利用する必要がある。

Android では Android KeyStore による暗号化が推奨されている。サンプルコードについては、以下のルールブックを参照。

データストレージとプライバシー要件ルールブック

- 暗号化キーの保存方法（必須）

これに違反する場合、以下の可能性がある。

- 脆弱性を含む実装となる可能性がある。

3.1.3.13 パディングオラクル攻撃に対抗するために、CBC を HMAC と組み合わせたりパディングエラー, MAC エラー, 復号失敗などのエラーが出ないようにする（必須）

CBC モードでは、平文ブロックは直前の暗号文ブロックと XOR される。これにより、ブロックに同じ情報が含まれている場合でも、暗号化された各ブロックは一意であり、ランダムであることが保証される。

CBC モード実装の例:

```
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)
    }
}
```

これに違反する場合、以下の可能性がある。

- パディングオラクル攻撃に対して脆弱になる。

3.1.3.14 暗号化されたデータを保存する場合、Galois/Counter Mode (GCM) のような、保存データの完全性も保護するブロックモードを使用する（推奨）

暗号化されたデータを保存する場合、Galois/Counter Mode (GCM) のような、保存データの完全性も保護するブロックモードを使用することを推奨する。後者には、このアルゴリズムが各 TLSv1.2 の実装に必須であり、したがってすべての最新のプラットフォームで利用できるという利点もある。

GCM モード実装の例:

```
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)
```

(次のページに続く)

(前のページからの続き)

```
        // 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
    }

}
```

これに違反する場合、以下の可能性がある。

- データのパターンを容易に特定される。

3.1.3.15 IV は暗号的に安全な乱数ジェネレーターを用いて生成する（必須）

CBC 、OFB 、CFB 、PCBC 、GCM モードでは、暗号への初期入力として、初期化ベクトル (IV) が必要である。IV は秘密にする必要はないが、予測可能であってはならない。暗号化されたメッセージごとにランダムで一意であり、非再現性である必要がある。そのため、IV は暗号的に安全な乱数ジェネレーターを用いて生成する必要がある。IV の詳細については、Crypto Fail の初期化ベクトルに関する記事を参照する。

サンプルコードは以下ルールブックを参照。

ルールブック

- セキュアな乱数ジェネレーターと設定を使用する（必須）

これに違反する場合、以下の可能性がある。

- 予測可能な初期化ベクトルが生成される。

3.1.3.16 初期化ベクトルがカウンターであることが多い CTR モードと GCM モード を使用する場合は、IV の使用方法が異なることに注意する（必須）

初期化ベクトルがカウンター (CTR と nonce の組み合わせ) であることが多い CTR モードと GCM モード を使用する場合は、IV の使用方法が異なることに注意する。そのため、独自のステートフルモデルを持つ予測可能な IV を使用することが必要である。CTR では、新しいブロック操作のたびに、新しい nonce とカウンターを入力として使用する。例: 5120 ビット長の平文の場合では、20 個のブロックがあるため、nonce とカウンターで構成される 20 個の入力ベクトルが必要である。一方 GCM では、暗号化操作ごとに IV を 1 つだけ持ち、同じキーで繰り返さないようにする。IV の詳細と推奨事項については、GCM に関する NIST の資料の 8 項を参照する。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- 各モードで必要とされる初期化ベクトルの要件を満たせない。

3.1.3.17 非対称暗号を行う際のパディングメカニズムとして PKCS#1 v2.0 に取り込まれた OAEP を使用する（必須）

以前は、非対称暗号を行う際のパディングメカニズムとして、PKCS1.5 パディング (コード: PKCS1Padding) が使われていた。このメカニズムは、パディングオラクル攻撃に対して脆弱である。そのため、PKCS#1 v2.0 (コード: OAEPwithSHA-256andMGF1Padding 、OAEPwithSHA-224andMGF1Padding 、OAEPwithSHA-384andMGF1Padding 、OAEPwithSHA-512andMGF1Padding) に取り込まれた OAEP (Optimal Asymmetric Encryption PaddingOAEPPadding) を使用するのが最適である。なお、OAEP を使用した場合でも、Kudelskisecurity のブログで紹介されている Mangers 攻撃としてよく知られている問題に遭遇する可能性があることに注意する。

以下のサンプルコードは、OAEP の使用方法である。

```
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))
        }
```

これに違反する場合、以下の可能性がある。

- パディングオラクル攻撃に対して脆弱になる。

3.1.3.18 メモリダンプを考慮してキーを使用する（必須）

メモリダンプが脅威モデルの一部である場合、キーがアクティブに使用された瞬間にキーにアクセスできる。メモリダンプには、ルートアクセス (ルート化されたデバイスやジェイルブレイクされたデバイスなど) が必要であるか、Frida でパッチされたアプリケーション (Fridump などのツールを使用できるように) が必要である。したがって、デバイスでキーがまだ必要な場合は、次のことを考慮するのが最善である。

- リモートサーバのキー: Amazon KMS や Azure Key Vault などのリモート Key Vault を使用することができる。一部のユースケースでは、アプリとリモートリソースの間にオーケストレーションレイヤーを開発することが適切なオプションとなる場合がある。
- ハードウェアで保護された安全なストレージ内のキー: すべての暗号化アクションとキー自体が信頼できる実行環境 (例: Android Keystore を使用) にあることを確認する。詳細については、Android Data Storage の章を参照する。
- エンベロープ暗号化によって保護されたキー: キーが TEE/SE の外部に保存されている場合は、multi-layered 暗号化の使用を検討する。エンベロープ暗号化アプローチ (OWASP Cryptographic Storage Cheat Sheet、Google Cloud Key management guide、AWS Well-Architected Framework guide 参照)、またはデータ暗号化鍵をキー暗号化する HPKE アプローチを使用する。
- メモリ内のキー: キーができるだけ短時間しかメモリに残さないようにし、暗号化操作に成功した後やエラー時にキーをゼロにし、無効化することを考慮する。一般的な暗号化のガイドラインについては、機密データのメモリの消去を参照する。より詳細な情報については、「機密データのメモリのテスト」を参照する。以下サンプルコードは、アプリでのメモリ内のキーの漏洩防止用の処理。

```
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())
    }
}
```

これに違反する場合、以下の可能性がある。

- メモリ内のキーが漏洩する可能性がある。

3.1.3.19 アカウントやデバイス間で同じキーを共有しない（必須）

メモリダンプが容易になるため、署名の検証や暗号化に使用される公開鍵以外は、アカウントやデバイス間で同じキーを共有しない。

※非推奨なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- キーのメモリダンプが容易になる。

3.1.3.20 トランスポート対称鍵や他のメカニズムによって、適切なキー保護を行う（必須）

キーをデバイス間で、またはアプリからバックエンドに転送する必要がある場合、トランスポート対称鍵や他のメカニズムによって、適切なキー保護が行われていることを確認する。多くの場合、キーは難読化された状態で共有されるため、簡単に元に戻すことができる。代わりに、非対称暗号化またはラッピングキーが使用されていることを確認する。例えば、対称鍵は非対称鍵の公開鍵で暗号化することができる。

キーを適切に保護するには KeyStore を使用する。KeyStore によるキーの保管方法は以下ルールブックを参照。

ルールブック

- キーがセキュリティハードウェアの内部に保存されているかどうかを確認する（推奨）

これに違反する場合、以下の可能性がある。

- キーを元に戻され読み取られる。

3.1.3.21 ハードコードされた対称暗号化を暗号化の唯一の方法として使用しない（必須）

暗号化の唯一の方法として、ハードコードされた対称暗号化を使用しないこと。以下は確認手順の一例。

1. 対称暗号化のすべてのインスタンスを識別する
2. 識別されたインスタンスごとに、ハードコードされた対称鍵があるかどうかを確認する。
3. ハードコードされた対称暗号化が暗号化の唯一の方法として使用されていないかどうかを確認する。

※デバッグ方法のため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- 暗号化方式が読み取られる。

3.1.3.22 暗号化メソッドのメソッドトレースを実施し、キーマテリアルの書き込み先または読み取り元を確認する（必須）

暗号化メソッドのメソッドトレースにより、使用されているキーなどの入出力値を把握することができる。暗号化操作の実行中にファイルシステムアクセスを監視して、キーマテリアルの書き込み先または読み取り元を評価する。例えば、RMS - Runtime Mobile Security の API モニターを使用して、ファイルシステムを監視することができる。

これに違反する場合、以下の可能性がある。

- 意図しない処理でキーマテリアルを使用される。

3.2 MSTG-CRYPTO-2

アプリは実績のある暗号化プリミティブの実装を使用している。

3.2.1 問題のある暗号化構成

※問題のある暗号化構成については、「MSTG-CRYPTO-1 3.1.1. 問題のある暗号化構成」の内容を確認すること。

3.2.2 暗号化標準アルゴリズムの構成

これらのテストケースは、暗号プリミティブの実装と使用に重点を置いている。以下のチェックを行うこと。

- 暗号プリミティブのすべてのインスタンスとその実装 (ライブラリまたはカスタム実装) を確認する。
- 暗号プリミティブの使用方法と設定方法を確認する。
- 使用されている暗号化プロトコルやアルゴリズムが、セキュリティ上、非推奨でないことを確認する。

参考資料

- owasp-mastg Testing the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4) Overview

ルールブック

- 適切な暗号化標準アルゴリズムの構成（必須）

3.2.2.1 静的解析

コード内の暗号化プリミティブのインスタンスをすべて特定する。すべてのカスタム暗号化の実装を特定する。

- クラス : Cipher 、Mac 、MessageDigest 、Signature
- インターフェース : Key 、PrivateKey 、PublicKey 、SecretKey
- 関数 : getInstance 、generateKey
- 例外 : KeyStoreException 、CertificateException 、NoSuchAlgorithmException
- java.security.* 、javax.crypto.* 、android.security.* 、android.security.keystore.* パッケージを使用するクラス

getInstance のすべての呼び出しが、プロバイダを指定しないことで、セキュリティサービスのデフォルトプロバイダを使用していることを確認する (AndroidOpenSSL 、別名 Conscrypt を意味する)。プロバイダは、KeyStore 関連のコードでのみ指定できる (その場合、KeyStore はプロバイダとして提供する必要がある)。他のプロバイダが指定されている場合は、状況とビジネスケース (Android API のバージョン) に従って検証する必要があり、潜在的な脆弱性に対してプロバイダを調べる必要がある。

「モバイルアプリの暗号化」の章で説明したベストプラクティスに従っていることを確認する。安全でない非推奨のアルゴリズムと一般的な構成の問題を調べる。

参考資料

- owasp-mastg Testing the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4) Static Analysis

ルールブック

- *getInstance* のすべての呼び出しで、セキュリティサービスのプロバイダを指定していないことを確認する（必須）

3.2.2.2 動的解析

※ MSTG-CRYPTO-1 へ同一内容を記載しているため、本章への記載を省略。

参考資料

- owasp-mastg Testing the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4) Dynamic Analysis

3.2.3 ルールブック

1. 適切な暗号化標準アルゴリズムの構成（必須）
2. *getInstance* のすべての呼び出しで、セキュリティサービスのプロバイダを指定していないことを確認する（必須）

3.2.3.1 適切な暗号化標準アルゴリズムの構成（必須）

暗号化プリミティブのインスタンスでの getInstance のすべての呼び出しが、プロバイダを指定しないことで、セキュリティサービスのデフォルトプロバイダを使用していることを確認する (AndroidOpenSSL 、別名 Conscrypt を意味する)。プロバイダは、KeyStore 関連のコードでのみ指定できる (その場合、KeyStore はプロバイダとして提供する必要がある)。他のプロバイダが指定されている場合は、状況とビジネスケース (Android API のバージョン) に従って検証する必要があり、潜在的な脆弱性に対してプロバイダを調べる必要がある。

以下は暗号化プリミティブに関係するキーワードの一例。

- クラス : Cipher 、Mac 、MessageDigest 、Signature
- インターフェース : Key 、PrivateKey 、PublicKey 、SecretKey
- 関数 : getInstance 、generateKey
- 例外 : KeyStoreException 、CertificateException 、NoSuchAlgorithmException
- java.security.* 、javax.crypto.* 、android.security.* 、android.security.keystore.*

また、セキュリティ上、非推奨でないことを確認する。

これに違反する場合、以下の可能性がある。

- 潜在的な脆弱性を含む暗号化アルゴリズムが使用される。

3.2.3.2 getInstance のすべての呼び出しで、セキュリティサービスのプロバイダを指定していないことを確認する（必須）

getInstance のすべての呼び出しで、セキュリティサービスのプロバイダを指定していないことを確認する。以下の方法で、プロバイダを調べる必要がある。

```
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(new KeyStore.LoadStoreParameter() {
   @Override
   public KeyStore.ProtectionParameter getProtectionParameter() {
      return null;
   }
});

Provider provider = keyStore.getProvider();
```

これに違反する場合、以下の可能性がある。

- 潜在的な脆弱性を含む暗号化アルゴリズムが使用される。

3.3 MSTG-CRYPTO-3

アプリは特定のユースケースに適した暗号化プリミティブを使用している。業界のベストプラクティスに基づくパラメータで構成されている。

3.3.1 問題のある暗号化構成

※問題のある暗号化構成については、「MSTG-CRYPTO-1 3.1.1. 問題のある暗号化構成」の内容を確認すること。

3.3.2 暗号化標準アルゴリズムの構成

※暗号化標準アルゴリズムの構成については、「MSTG-CRYPTO-2 3.2.2. 暗号化標準アルゴリズムの構成」の内容を確認すること。

3.4 MSTG-CRYPTO-4

アプリはセキュリティ上の目的で広く非推奨と考えられる暗号プロトコルやアルゴリズムを使用していない。

3.4.1 セキュアでない、または非推奨な暗号化アルゴリズム

モバイルアプリを評価する際には、重大な既知の弱点を持つ暗号アルゴリズムやプロトコルを使用していないこと、あるいは最新のセキュリティ要件に対して不十分な点がないことを確認する必要がある。過去に安全とされていたアルゴリズムも、時間の経過とともに安全でなくなる可能性がある。したがって、現在のベストプラクティスを定期的にチェックし、それに応じて設定を調整することが重要である。

暗号化アルゴリズムが最新のものであり、業界標準に準拠していることを確認する。脆弱なアルゴリズムには、旧式のブロック暗号 (DES 、3DES など) 、ストリーム暗号 (RC4 など) 、ハッシュ関数 (MD5 、SHA1 など) 、壊れた乱数ジェネレーター (Dual_EC_DRBG 、SHA1PRNG など) が含まれる。認証されているアルゴリズム (NIST など) でも、時間の経過とともに安全でなくなる可能性があることに注意する。認証は、アルゴリズムの健全性を定期的に検証することに取って代わるものではない。既知の弱点を持つアルゴリズムは、より安全な代替手段に置き換える必要がある。さらに、暗号化に使用されるアルゴリズムは標準化され、検証可能である必要がある。未知のアルゴリズムや独自のアルゴリズムを使ってデータを暗号化すると、アプリケーションがさまざまな暗号攻撃にさらされ、平文が復元される可能性がある。

アプリのソースコードを調査し、以下のような脆弱性が知られている暗号化アルゴリズムのインスタンスを特定する。

- DES,3DES
- RC2
- RC4
- BLOWFISH
- MD4
- MD5
- SHA1

暗号化 API の名前は、特定のモバイルプラットフォームによって異なる。

次のことを確認する。

- 暗号化アルゴリズムは最新で、業界標準に準拠している。これには、時代遅れのブロック暗号 (DES など) ストリーム暗号 (RC4 など)、ハッシュ関数 (MD5 など)、Dual_EC_DRBG などの破損した乱数ジェネレーター (NIST 認定であっても) が含まれますが、これらに限定されない。これらはすべて安全でないものとしてマークし、使用せず、アプリケーションとサーバから削除する必要がある。
- キーの長さは業界標準に準拠しており、十分な時間の保護を提供する。ムーアの法則を考慮したさまざまなキーの長さとその保護性能の比較は、オンラインで確認可能である。

- 暗号化手段は互いに混合されていない: 例えば、公開鍵で署名したり、署名に使用した対称鍵を暗号化に再利用しない。
- 暗号化パラメータが合理的な範囲で適切に定義されている。これには、ハッシュ関数出力と少なくとも同じ長さである必要がある暗号ソルト、パスワード導出関数と反復回数の適切な選択 (例: PBKDF2 、scrypt 、bcrypt) 、IV はランダムでユニークであること、目的に合ったブロック暗号化モード (例: ECB は特定の場合を除き使用しない)、キー管理が適切に行われているか (例: 3DES は 3 つの独立したキーを持つべきである) などが含まれるが、これらに限定されない。

参考資料

- owasp-mastg Identifying Insecure and/or Deprecated Cryptographic Algorithms (MSTG-CRYPTO-4)

ルールブック

- **セキュアでない、または非推奨な暗号化アルゴリズムは使用しない（必須）**

3.4.2 暗号化標準アルゴリズムの構成

※暗号化標準アルゴリズムの構成については、「MSTG-CRYPTO-2 3.2.2. 暗号化標準アルゴリズムの構成」の内容を確認すること。

3.4.3 ルールブック

1. **セキュアでない、または非推奨な暗号化アルゴリズムは使用しない（必須）**

3.4.3.1 セキュアでない、または非推奨な暗号化アルゴリズムは使用しない（必須）

業界標準に準拠した最新の暗号化アルゴリズムで実装すること。

具体的な観点としては以下内容に準拠して実装する。

- 暗号化アルゴリズムは最新で、業界標準に準拠している。業界標準については「**業界標準を満たしたキーの長さを設定する（必須）**」を参照。
- キーの長さは業界標準に準拠し、十分な時間の保護を提供する。ムーアの法則を考慮したさまざまなキーの長さとその保護性能の比較は、オンラインで確認可能である。
- 暗号化手段を互いに混合しない: 例えば、公開鍵で署名したり、署名に使用した対称鍵を暗号化に再利用しない。
- 暗号化パラメータを合理的な範囲で適切に定義する。これには、ハッシュ関数出力と少なくとも同じ長さである必要がある暗号ソルト、パスワード導出関数と反復回数の適切な選択 (例: PBKDF2 、scrypt 、bcrypt) 、IV はランダムでユニークであること、目的に合ったブロック暗号化モード (例: ECB は特定の場合を除き使用しない)、キー管理が適切に行われているか (例: 3DES は 3 つの独立したキーを持つべきである) などが含まれるが、これらに限定されない。

サンプルコードは、以下ルールブックを参照。

ルールブック

- 目的に応じた暗号化を使用する（必須）

これに違反する場合、以下の可能性がある。

- 脆弱な暗号化処理となる可能性がある。

3.5 MSTG-CRYPTO-5

アプリは複数の目的のために同じ暗号化鍵を再利用していない。

3.5.1 暗号化鍵の利用目的と再利用の検証

このテストケースは、利用目的の検証と同じ暗号キーの再利用に焦点を合わせている。以下のチェックを実行する必要がある。

- 暗号化が使用されているすべてのインスタンスを特定する
- 暗号素材の目的を特定する (使用中、転送中、保存中のデータを保護する)
- 暗号の種類を識別する
- 暗号が目的に応じて使用されているかを確認する

3.5.1.1 静的解析

暗号化が使用されているすべてのインスタンスを確認する。

- クラス: Cipher 、Mac 、MessageDigest 、Signature
- インターフェース: Key 、PrivateKey 、PublicKey 、SecretKey
- 関数: getInstance 、generateKey
- 例外: KeyStoreException 、CertificateException 、NoSuchAlgorithmException
- インポートクラス: java.security.* 、javax.crypto.* 、android.security.* and android.security.keystore.*

特定された各インスタンスについて、その目的と種類を特定し使用することが可能である。

- 暗号化/復号 - データの機密性を確保するため。
- signing/verifying - データの完全性を確保するため (場合によっては説明責任も)。
- メンテナンス - 特定の機密性の高い操作 (KeyStore へのインポートなど) の際にキーを保護するため。

さらに、特定された暗号化のインスタンスを使用するビジネスロジックを特定する必要がある。

検証の際には、以下のチェックを行う必要がある。

- すべてのキーが作成時に定義された目的に従って使用されているか (KeyProperties を定義できる KeyStore のキーに関連する) 。
- 非対称鍵の場合、秘密鍵は署名に、公開鍵は暗号化にのみ使用されているか。
- 対称鍵は複数の目的に使用されているか。異なるコンテキストで使用される場合は、新しい対称鍵を生成する必要がある。
- 暗号化はビジネス上の目的に応じて使用されているか。

参考資料

- owasp-mastg Testing the Purposes of Keys (MSTG-CRYPTO-5) Static Analysis

ルールブック

- 目的に応じた暗号化を使用する（必須）

3.5.1.2 動的解析

※暗号化鍵の利用目的と再利用の検証の動的解析については、「MSTG-CRYPTO-1 3.1.2.2. 動的解析」の内容を確認すること。

参考資料

- owasp-mastg Testing the Purposes of Keys (MSTG-CRYPTO-5) Dynamic Analysis

3.5.2 ルールブック

1. 目的に応じた暗号化を使用する（必須）

3.5.2.1 目的に応じた暗号化を使用する（必須）

目的に応じた暗号化を使用すること。

具体的な観点としては以下内容に準拠すること。

- すべてのキーが作成時に定義された目的に従って使用する (KeyProperties を定義できる KeyStore のキーに関連する) 。
- 非対称鍵の場合、秘密鍵は署名に、公開鍵は暗号化にのみ使用する。
- 対称鍵は複数の目的に使用しない。異なるコンテキストで使用される場合は、新しい対称鍵を生成する必要がある。
- 暗号化はビジネス上の目的に応じて使用する。

以下はソースコード内での暗号化に関係するキーワードの一例。

- クラス : Cipher 、Mac 、MessageDigest 、Signature

- インターフェース : Key 、PrivateKey 、PublicKey 、SecretKey
- 関数 : getInstance 、generateKey
- 例外 : KeyStoreException 、CertificateException 、NoSuchAlgorithmException
- java.security.* 、javax.crypto.* 、android.security.* 、android.security.keystore.*

これに違反する場合、以下の可能性がある。

- 脆弱な暗号化処理となる可能性がある。

3.6 MSTG-CRYPTO-6

すべての乱数値は十分にセキュアな乱数ジェネレーターを用いて生成されている。

3.6.1 乱数ジェネレーターの選択

このテストケースは、アプリケーションで使用される乱数値に焦点を合わせている。以下のチェックを実行する必要がある。

- 乱数値が使用されるすべてのインスタンスを特定する。
- 乱数ジェネレーターが暗号的に安全であるとみなされていないかどうかを確認する。
- 乱数ジェネレーターがどのように使用されるかを検証する。
- 生成された乱数値のランダム性を検証する。

乱数ジェネレーターのインスタンスをすべて特定し、カスタムクラスまたは既知の安全でないクラスを探す。例えば、java.util.Random は与えられたシード値に対して同一の数値列を生成する。その結果、数値列は予測可能となる。その代わりに、その分野の専門家が現在強力だと考えているよく吟味されたアルゴリズムを選ぶべきで、適切な長さのシードでよくテストされた実装を使用する必要がある。

参考資料

- owasp-mastg Testing Random Number Generation (MSTG-CRYPTO-6) Overview

ルールブック

- セキュアな乱数ジェネレーターと設定を使用する（必須）

3.6.1.1 セキュアな乱数ジェネレーター

デフォルトコンストラクタで生成されない SecureRandom のインスタンスをすべて特定する。シード値を指定すると、ランダム性が低下する場合がある。システムで指定されたシード値を使用して 128 バイト長の乱数を生成する SecureRandom の引数なしのコンストラクタを優先する。

参考資料

- owasp-mastg Testing Random Number Generation (MSTG-CRYPTO-6) Static Analysis

ルールブック

- **セキュアな乱数ジェネレーターと設定を使用する（必須）**

3.6.1.2 セキュアでない乱数ジェネレーター

一般的に、PRNG が暗号的に安全であると宣伝されていない場合 (例えば、java.util.Random)、それはおそらく統計的 PRNG であり、セキュリティが重視されるコンテキストでは使用すべきではない。擬似乱数ジェネレーターは、ジェネレーターが既知でシードを推測できる場合、予測可能な数値を生成できる。128 ビットのシードは、「十分な乱数」を生成するための出発点として適している。

攻撃者が使用されている弱い擬似乱数ジェネレーター (PRNG) のタイプを知れば、Java Random で行われたように、以前に観測された値に基づいて次の乱数値を生成する概念実証を作成するのは簡単である。非常に弱いカスタム乱数ジェネレーターの場合、パターンを統計的に観察できる場合がある。ただし、推奨されるアプローチは、APK を逆コンパイルしてアルゴリズムを検査することである。

ランダム性をテストしたい場合は、大きな数値のセットを取得して Burp's sequencer でランダム性の品質を確認することができる。

上記のクラスとメソッドのメソッドトレースを使用して、使用されている入力/出力値を特定できる。

参考資料

- owasp-mastg Testing Random Number Generation (MSTG-CRYPTO-6) Static Analysis
- owasp-mastg Testing Random Number Generation (MSTG-CRYPTO-6) Dynamic Analysis

ルールブック

- **セキュアな乱数ジェネレーターと設定を使用する（必須）**

3.6.2 ルールブック

1. **セキュアな乱数ジェネレーターと設定を使用する（必須）**

3.6.2.1 セキュアな乱数ジェネレーターと設定を使用する（必須）

乱数を使用する場合、乱数のランダム性により安全性が変わる。安全性を高めるためには、よりセキュアな乱数ジェネレーターと設定により、乱数を生成する必要がある。

以下はセキュアな乱数ジェネレーターの一例。

- SecureRandom

SecureRandom を利用する場合は、引数なしのコンストラクタにより、インスタンスを生成する。シード値を指定してインスタンスを生成した場合、ランダム性が低下する場合がある。そのため、システムにより指定されたシード値を使用して 128 バイト長の乱数を生成する。

以下ソースコードは SecureRandom の引数なしのコンストラクタによるインスタンス生成例。

```
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));
  }
}
```

一方、セキュアでない乱数ジェネレーターには以下のものがある。

- java.util.Random

上記は擬似乱数ジェネレーター（ PRNG ）であり、セキュリティが重視されるコンテキストでは使用すべきではない。擬似乱数ジェネレーターは、ジェネレーターが既知でシード値を推測できる場合、予測可能な数値を生成できる。

以下ソースコードは java.util.Random による乱数の生成例。

```
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);
}
```

これに違反する場合、以下の可能性がある。

- 安全性の低い乱数が生成される。

4 認証とセッション管理要件

4.1 MSTG-AUTH-1

アプリがユーザにリモートサービスへのアクセスを提供する場合、ユーザ名/パスワード認証など何らかの形態の認証がリモートエンドポイントで実行されている。

※本章での適切な認証対応については「*MSTG-ARCH-2 の 1.2.1.1. 適切な認証対応*」を参照する。

参考資料

- 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 認証情報の確認

資格情報の確認フローは Android 6.0 以降で利用可能であり、ユーザがロック画面保護と共にアプリ固有のパスワードを入力する必要がないようにするために使用される。代わりに、ユーザが最近デバイスにログインした場合、confirm-credentials を使用して、AndroidKeystore から暗号化マテリアルのロックを解除できる。つまり、ユーザが設定された時間制限 (setUserAuthenticationValidityDurationSeconds) 内にデバイスのロックを解除した場合、それ以外の場合はデバイスを再度ロック解除する必要がある。なお、setUserAuthenticationValidityDurationSeconds は API level 30 で非推奨となっているため、現在は setUserAuthenticationParameters が推奨されている。

資格情報の確認のセキュリティは、ロック画面で設定された保護と同じくらい強力であることに注意する。これは多くの場合、単純な予測ロック画面パターンが使用されることを意味するため、認証情報の確認を使用するためにセキュリティ制御の L2 を必要とするアプリは推奨されない。

参考資料

- owasp-mastg Testing Confirm Credentials (MSTG-AUTH-1 and MSTG-STORAGE-11) Overview

ルールブック

- **資格情報の確認フローは、セキュリティ強度について理解して使用する（推奨）**

4.1.1.1 静的解析

ロック画面が設定されていることを確認する。

```
KeyguardManager mKeyguardManager = (KeyguardManager) getSystemService(Context.
↪KEYGUARD_SERVICE);
if (!mKeyguardManager.isKeyguardSecure()) {
    // Show a message that the user hasn't set up a lock screen.
}
```

- ロック画面で保護されたキーを作成する。このキーを使用するには、ユーザが最後の X 秒以内にデバイスのロックを解除するか、デバイスのロックを再度解除する必要がある。デバイスのロックを解除しているユーザとアプリを使用しているユーザが同じであることを確認するのが難しくなるため、このタイムアウトが長すぎないことを確認すること。

```
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);
}
```

- ロック画面をセットアップして確認する。

```
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);
}
```

なお、startActivityForResult は現在非推奨のため、推奨方法はルールブックを参照。

- ロック画面の後にキーを使用する。

```
@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.
        }
    }
}
```

なお、onActivityResult は現在非推奨のため、推奨方法はルールブックを参照。

アプリケーションフロー中にロック解除されたキーが使用されていることを確認する。たとえば、キーを使用して、ローカルストレージやリモートエンドポイントから受信したメッセージを復号できる。ユーザがキーのロックを解除したかどうかをアプリケーションが単純にチェックする場合、アプリケーションはローカル認証バイパスに対して脆弱である可能性がある。

参考資料

- owasp-mastg Testing Confirm Credentials (MSTG-AUTH-1 and MSTG-STORAGE-11) Static Analysis

ルールブック

- **資格情報の確認フローは、セキュリティ強度について理解して使用する（推奨）**

4.1.1.2 動的解析

ユーザが正常に認証された後、キーの使用が承認される期間 (秒) を検証する。これは、setUserAuthenticationRequired が使用されている場合にのみ必要である。

参考資料

- owasp-mastg Testing Confirm Credentials (MSTG-AUTH-1 and MSTG-STORAGE-11) Dynamic Analysis

ルールブック

- **資格情報の確認フローは、セキュリティ強度について理解して使用する（推奨）**

4.1.2 ルールブック

1. 資格情報の確認フローは、セキュリティ強度について理解して使用する（推奨）

4.1.2.1 資格情報の確認フローは、セキュリティ強度について理解して使用する（推奨）

資格情報の確認のセキュリティは、ロック画面で設定された保護と同じくらい強力であることに注意する。これは多くの場合、単純な予測ロック画面パターンが使用されることを意味するため、認証情報の確認を使用するためにセキュリティ制御の L2 を必要とするアプリは推奨されない。

以下へ資格情報の確認フローのサンプルコードを示す。

```
    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) {
                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
```

(次のページに続く)

(前のページからの続き)

```
                // 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
                    public void onAuthenticationSucceeded(BiometricPrompt.
↪AuthenticationResult result) {
                        super.onAuthenticationSucceeded(result);
                        // Authentication Success
                    }

                    @Override
                    public void onAuthenticationFailed() {
                        super.onAuthenticationFailed();
                        // Authentication failure
                    }
```

(次のページに続く)

(前のページからの続き)

```
                };
                biometricPrompt.authenticate(cancellationSignal, executors,␣
↪authCallBack);
            } else {
                Intent intent = keyguardManager.
↪createConfirmDeviceCredentialIntent(null, null);
                checkCredentialLauncher.launch(intent);
            }
        }
    }
```

これに注意しない場合、以下の可能性がある。

- 強度の低いセキュリティでの資格情報の確認フローを実施する可能性がある。

4.2 MSTG-AUTH-2

ステートフルなセッション管理を使用する場合、リモートエンドポイントはランダムに生成されたセッション識別子を使用し、ユーザの資格情報を送信せずにクライアント要求を認証している。

4.2.1 セッション情報の管理

ステートフル (または「セッションベース」) 認証は、クライアントとサーバの両方に認証レコードがあることが特徴である。認証の流れは以下の通りである。

1. アプリはユーザの認証情報を含む要求をバックエンドサーバに送信する。
2. サーバは認証情報を確認する。認証情報が有効な場合、サーバはランダムなセッション ID とともに新しいセッションを作成する。
3. サーバはセッション ID を含む応答をクライアントに送信する。
4. クライアントは以降のすべての要求でセッション ID を送信する。サーバはセッション ID を検証し、関連するセッションレコードを取得する。
5. ユーザがログアウトした後、サーバ側のセッションレコードは破棄され、クライアントはセッション ID を破棄する。

セッションの管理が不適切な場合、様々な攻撃に対して脆弱となり、正規のユーザのセッションが侵害され、攻撃者がそのユーザになりすますことが可能になる。その結果、データの損失、機密性の低下、不正な操作が生じる可能性がある。

参考資料

- owasp-mastg Testing Stateful Session Management (MSTG-AUTH-2)

4.2.1.1 セッション管理のベストプラクティス

機密情報や機能を提供するサーバ側のエンドポイントを探し出し、一貫した認可の実施を検証する。バックエンドサービスは、ユーザのセッション ID またはトークンを検証し、ユーザがリソースにアクセスするための十分な権限を持っていることを確認する必要がある。セッション ID またはトークンがない、または無効な場合、要求を拒否する必要がある。

確認事項

- セッション ID は、サーバ側でランダムに生成される。
- ID は簡単に推測できないようにする。（適切な長さとエントロピーを使用する）
- セッション ID は常に安全な接続（例 : HTTPS ）で交換される。
- モバイルアプリはセッション ID を永続的なストレージに保存しない。
- サーバは、ユーザが権限を要するアプリケーションにアクセスするたびに、セッションを検証する。（セッション ID は有効で、適切な認証レベルに対応している必要がある）
- セッションはサーバ側で終了し、タイムアウトまたはユーザがログアウトした後にモバイルアプリ内でセッション情報が削除される。

認証は実装するべきでなく、実績のあるフレームワークの上に構築する必要がある。多くの一般的なフレームワークは、既製の認証およびセッション管理機能を提供している。アプリが認証のためにフレームワークの API を使用する場合、ベストプラクティスのためにフレームワークのセキュリティドキュメントを確認する。一般的なフレームワークのセキュリティガイドは、以下のリンクで入手できる。

- Spring (Java)
- Struts (Java)
- Laravel (PHP)
- Ruby on Rails

サーバ側の認証を検証するための最適なリソースは、OWASP Web Testing Guide 、特に、「Testing Authentication 」と「 Testing Session Management 」の章に掲載されている。

参考資料

- owasp-mastg Testing Stateful Session Management (MSTG-AUTH-2) Session Management Best Practices

ルールブック

- **モバイルアプリはセッション *ID* を永続的なストレージに保存しないことを確認する（必須）**

4.2.2 ルールブック

1. モバイルアプリはセッション *ID* を永続的なストレージに保存しないことを確認する（必須）

4.2.2.1 モバイルアプリはセッション ID を永続的なストレージに保存しないことを確認する（必須）

永続的なストレージにセッション ID を保存すると、ユーザにより読み書きされたり、第三者により利用されたりする恐れがある。そのため、セッション ID はこのようなストレージには保存しないようにする必要がある。

※非推奨なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- セッション ID をユーザにより読み書きされたり、第三者により利用されたりする恐れがある。

4.3 MSTG-AUTH-3

ステートレスなトークンベースの認証を使用する場合、サーバはセキュアなアルゴリズムを使用して署名されたトークンを提供している。

4.3.1 トークンの管理

トークンベースの認証は、HTTP 要求ごとに署名されたトークン（サーバによって検証される）を送信することによって実装される。最も一般的に使用されているトークン形式は、RFC7519 で定義されている JSON Web Token である。JWT は、完全なセッション状態を JSON オブジェクトとしてエンコードすることができる。そのため、サーバはセッションデータや認証情報を保存する必要がない。

JWT トークンは、ドットで区切られた 3 つの Base64Url エンコード部分から構成されている。トークンの構造は以下の通りである。

```
base64UrlEncode(header).base64UrlEncode(payload).base64UrlEncode(signature)
```

次の例は、Base64Url でエンコードされた JSON Web Token を示している。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

ヘッダーは通常 2 つの部分から構成される。トークンタイプ（ JWT ）と、署名を計算するために使用されるハッシュアルゴリズムである。上の例では、ヘッダーは以下のようにデコードされる。

```
{"alg":"HS256","typ":"JWT"}
```

トークンの 2 番目の部分はペイロードで、いわゆるクレームを含んでいる。クレームは、あるエンティティ（通常はユーザ）に関するステートメントと、追加のメタデータである。例を以下に示す。

```
{"sub":"1234567890","name":"John Doe","admin":true}
```

署名は、符号化されたヘッダー、符号化されたペイロード、および秘密値に対して、JWT ヘッダーで指定されたアルゴリズムを適用することによって作成される。例えば、HMAC SHA256 アルゴリズムを使用する場合、署名は以下のように作成される。

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

この秘密情報は、認証サーバとバックエンドサービスの間で共有され、クライアントはそれを認識しないことに注意する。これは、トークンが正当な認証サービスから取得されたことを証明するものである。また、クライアントがトークンに含まれるクレームを改ざんすることも防ぐことができる。

参考資料

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3)

4.3.2 静的解析

サーバとクライアントが使用している JWT ライブラリを特定する。使用している JWT ライブラリに既知の脆弱性があるかどうかを確認する。

実装が JWT の ベストプラクティス を遵守していることを確認する。

- トークンを含むすべての受信要求について、HMAC が検証されることを確認する。
- 秘密署名鍵または HMAC 秘密鍵の場所を確認する。キーはサーバに保管し、決してクライアントと共有してはならない。発行者と検証者のみが利用可能であるべきである。
- JWT に個人を特定できる情報のような機密データが埋め込まれていないことを確認する。何らかの理由でトークン内にそのような情報を送信する必要があるアーキテクチャの場合、ペイロードの暗号化が適用されていることを確認する。OWASP JWT Cheat Sheet にある Java 実装のサンプルを参照する。
- JWT に一意な識別子を与える jti (JWT ID) クレームでリプレイ攻撃に対処していることを確認する。
- クロスサービスリレー攻撃には、トークンがどのアプリケーションのためのものかを定義する aud (audience) クレームで対処していることを確認する。
- トークンは、KeyStore (Android) により、モバイルデバイス上に安全に保管されていることを確認する。

参考資料

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Static Analysis

ルールブック

- **使用している *JWT* ライブラリの特定及び既知の脆弱性があるかどうかを確認する（必須）**
- **トークンは、*KeyStore (Android)* により、モバイルデバイス上に安全に保管されていることを確認する（必須）**

4.3.2.1 ハッシュアルゴリズムの強制

攻撃者はトークンを変更し、「 none 」キーワードを使って署名アルゴリズムを変更し、トークンの完全性がすでに検証されていることを示すことによってこれを実行する。ライブラリによっては、「 none 」アルゴリズムで署名されたトークンを、検証済みの署名を持つ有効なトークンであるかのように扱い、アプリケーションは変更されたトークンの要求を信頼する可能性がある。

例えば、Java アプリケーションでは、検証コンテキストを作成する際に、期待するアルゴリズムを明示的に要求する必要がある。

```
// 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);
```

参考資料

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Enforcing the Hashing Algorithm

ルールブック

- **期待するハッシュアルゴリズムを要求する必要がある（必須）**

4.3.2.2 トークンの有効期限

署名されたステートレス認証トークンは、署名キーが変更されない限り永久に有効である。トークンの有効期限を制限する一般的な方法は、有効期限を設定することである。トークンに "exp" expiration claim が含まれ、バックエンドが期限切れのトークンを処理しないことを確認する。

トークンを付与する一般的な方法は、アクセストークンとリフレッシュトークンを組み合わせることである。ユーザがログインすると、バックエンドサービスは有効期間の短いアクセストークンと有効期間の長いリフレッシュトークンを発行する。アクセストークンの有効期限が切れた場合、アプリケーションはリフレッシュトークンを使って新しいアクセストークンを取得することができる。

機密データを扱うアプリの場合、リフレッシュトークンの有効期限が適切な期間であることを確認する必要がある。次のコード例では、リフレッシュトークンの発行日をチェックするリフレッシュトークン API を紹介する。トークンが 14 日以上経過していない場合、新しいアクセストークンが発行される。それ以外の場合は、アクセスが拒否され、再度ログインするよう促される。

```
app.post('/renew_access_token', function (req, res) {
    // verify the existing refresh token
```

(次のページに続く)

(前のページからの続き)

```
    var profile = jwt.verify(req.body.token, secret);

    // 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 });
  });
```

参考資料

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Token Expiration

ルールブック

- **リフレッシュトークンの有効期限に適切な期間を設定する（必須）**

4.3.3 動的解析

動的解析を行いながら、以下の JWT の脆弱性を調査する。

- クライアント上のトークン保存場所
  - JWT を使用するモバイルアプリの場合、トークンの保管場所を確認する必要がある。
- 署名キーのクラック
  - トークン署名は、サーバ上の秘密キーを介して作成される。JWT を取得した後、オフラインで秘密鍵をブルートフォースするためのツールを選択する。
- 情報の開示
  - Base64Url でエンコードされた JWT をデコードし、それがどのようなデータを送信しているか、そのデータが暗号化されているかどうかを調べることができる。
- ハッシュアルゴリズムの改ざん
  - 非対称型アルゴリズムの使用。JWT は RSA や ECDSA のような非対称アルゴリズムをいくつか提供している。これらのアルゴリズムが使用される場合、トークンは秘密鍵で署名され、検証には公開鍵が使用される。もしサーバが非対称アルゴリズムで署名されたトークンを期待し、HMAC で署名されたトークンを受け取った場合、サーバは公開鍵を HMAC 秘密鍵として扱う。そして、公開鍵はトークンに署名するために HMAC 秘密鍵として採用され、悪用される可能性がある。

  - トークンヘッダの alg 属性を修正し、HS256 を削除して none に設定し、空の署名 (例えば、signature = "") を使用する。このトークンを使用し、要求で再生する。いくつかのライブラリは、none アルゴリズムで署名されたトークンを、検証済みの署名を持つ有効なトークンとして扱う。これにより、攻撃者は自分自身の「署名付き」トークンを作成することができる。

上記のような脆弱性をテストするために、2 種類の Burp プラグインが用意されている。

- JSON Web Token Attacker
- JSON Web Tokens

また、その他の情報については、OWASP JWT Cheat Sheet を必ず確認すること。

参考資料

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Dynamic Analysis

4.3.4 OAuth 2.0 のフローテスト

OAuth 2.0 は、API やウェブ対応アプリケーションのネットワーク上で認可の決定を伝えるための委任プロトコルを定義している。ユーザ認証アプリケーションなど、さまざまなアプリケーションで使用されている。

OAuth2 の一般的な用途は以下の通りである。

- ユーザのアカウントを使用してオンラインサービスにアクセスする許可をユーザから取得する。
- ユーザに代わってオンラインサービスに対する認証を行う。
- 認証エラーの処理。

OAuth 2.0 によると、ユーザのリソースにアクセスしようとするモバイルクライアントは、まずユーザに認証サーバに対する認証を要求する必要がある。ユーザの承認後、認証サーバはユーザに代わってアプリが動作することを許可するトークンを発行する。OAuth2 仕様では、特定の種類の認証やアクセストークンの形式は定義されていないことに注意する。

OAuth 2.0 では、4 つの役割が定義されている。

- リソースオーナー：アカウントの所有者
- クライアント：アクセストークンを使ってユーザのアカウントにアクセスしようとするアプリケーション
- リソースサーバ：ユーザアカウントをホストする
- 認可サーバ：ユーザの身元を確認し、アプリケーションにアクセストークンを発行する

注：API はリソースオーナーと認可サーバの両方の役割を果たす。したがって、ここでは両方を API と呼ぶことにする。

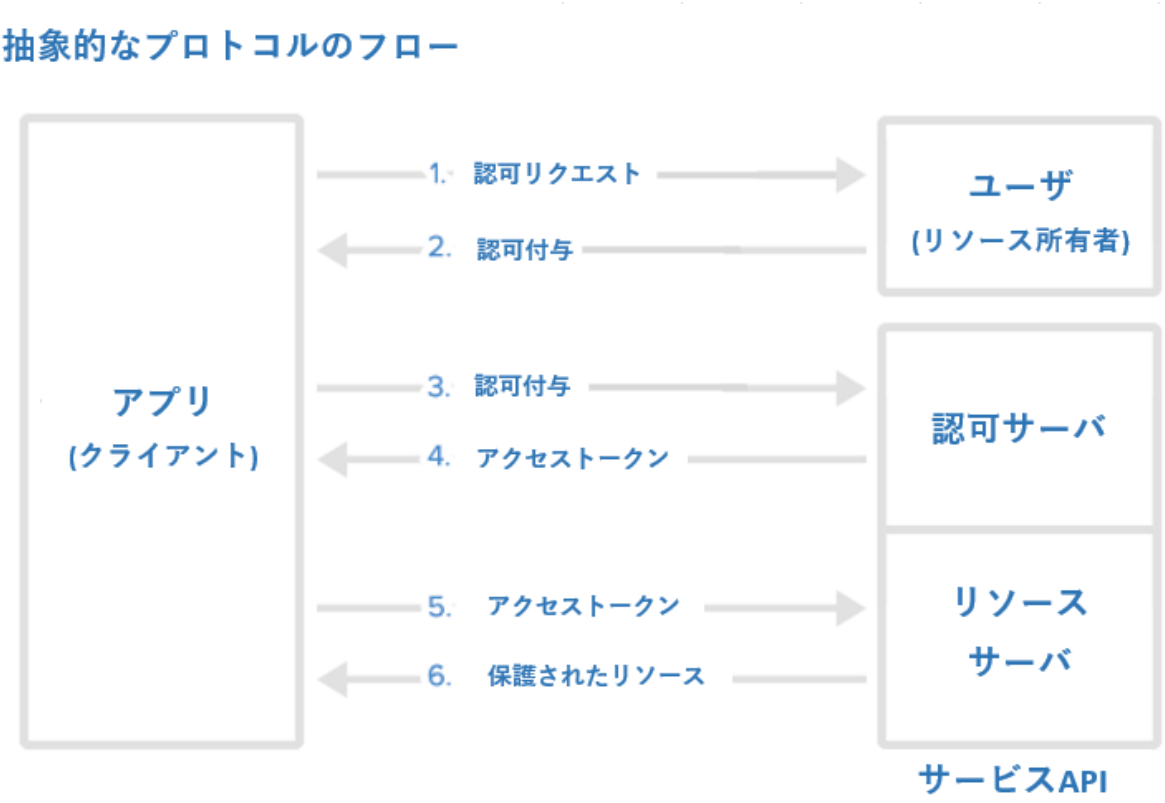


図 4.3.4.1 抽象的なプロトコルのフロー

図中の各ステップについて、以下でより詳細に説明する。

1. アプリケーションは、サービス・リソースにアクセスするためのユーザ認証を要求する。
2. ユーザが要求を承認すると、アプリケーションは認証付与を受け取る。認可付与は、いくつかの形式（明示的、暗黙的など）を取ることができる。
3. アプリケーションは、認可サーバ（API）にアクセストークンを要求する際に、認証付与と一緒に自身の ID の認証を提示する。
4. アプリケーションの ID が認証され、認証付与が有効である場合、認可サーバ（ API ）はアプリケーションにアクセストークンを発行し、認可プロセスを完了する。アクセストークンには、リフレッシュトークンが付随している場合がある。
5. アプリケーションはリソースサーバ（ API ）にリソースを要求し、認証のためにアクセストークンを提示する。アクセストークンは、いくつかの方法で使用することができる（ベアラートークンなど）。
6. アクセストークンが有効であれば、リソースサーバ（ API ）はアプリケーションにリソースを提供する。

参考資料

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-1 and MSTG-AUTH-3)

4.3.5 OAuth 2.0 のベストプラクティス

以下のベストプラクティスが守られていることを確認する。

ユーザエージェント

- ユーザは信頼を視覚的に確認する方法（例：TLS(Transport Layer Security) 確認、ウェブサイトの仕組み）を持つべきである。
- 中間者攻撃を防ぐために、クライアントはサーバの完全修飾ドメイン名を、接続確立時にサーバが提示した公開鍵で検証する必要がある。

付与の種類

- ネイティブアプリでは、暗黙的な付与ではなく、コード付与を使用する必要がある。
- コード付与を使用する場合、コード付与を保護するために PKCE (Proof Key for Code Exchange) を実装する必要があります。サーバ側でも実装されていることを確認する。
- 認可「コード」は有効期限が短く、受信後すぐに使用されるべきである。認可コードが一時的なメモリ上にのみ存在し、保存やログに記録されないことを確認する。

クライアントシークレット

- クライアントはなりすまされる可能性があるため、クライアントの ID を証明するために共有シークレットを使うべきではない（「 client_id 」がすでに証明の役目を果たしている）。もし、クライアントシークレットを使用する場合は、安全なローカルストレージに保存されていることを確認する。

エンドユーザの認証情報

- TLS などのトランスポート層方式でエンドユーザ認証情報の送信を保護する。

トークン

- アクセストークンは一時的なメモリに保存する。
- アクセストークンは、暗号化された接続で送信する必要がある。
- エンドツーエンドの機密性が保証されない場合や、トークンが機密情報や取引へのアクセスを提供する場合は、アクセストークンの範囲と期間を縮小する。
- アプリケーションがアクセストークンをベアラートークンとして使用し、クライアントを識別する他の方法がない場合、トークンを盗んだ攻撃者はそのスコープとそれに関連するすべてのリソースにアクセスできることに留意する。
- リフレッシュトークンは、安全なローカルストレージに保管する。

参考資料

- owasp-mastg Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3) OAUTH 2.0 Best Practices

ルールブック

- *OAuth 2.0* のベストプラクティスの遵守（必須）

4.3.5.1 外部ユーザエージェントと組み込みユーザエージェント

OAuth2 認証は、外部のユーザエージェント（ Chrome や Safari など）を介して行うことも、アプリ自体（アプリに埋め込まれた WebView や認証ライブラリなど）を介して行うこともできる。どちらのモードが本質的に「優れている」ということはなく、どのモードを選択するかはコンテキストに依存する。

外部ユーザエージェントを使用する方法は、ソーシャルメディアアカウント（ Facebook 、Twitter など）と対話する必要があるアプリケーションで選択される方法である。この方法のメリットは以下の通りである。

- ユーザの認証情報は、決してアプリに直接公開されることはない。このため、ログイン時にアプリが認証情報を取得することはできない（「認証情報のフィッシング」）。
- アプリ自体に認証ロジックをほとんど追加する必要がないため、コーディングミスを防ぐことができる。

マイナス面としては、ブラウザの動作を制御する方法（証明書のピン留めを有効にする等）がないことである。

閉じたエコシステムの中で動作するアプリの場合、埋め込み認証の方が良い選択となる。例えば、OAuth2 を使って銀行の認証サーバからアクセストークンを取得し、そのアクセストークンを使って多くのマイクロサービスにアクセスする銀行アプリを考える。この場合、クレデンシャルフィッシングは実行可能なシナリオではない。認証プロセスは、外部のコンポーネントを信頼するのではなく、（できれば）慎重に保護されたバンキングアプリの中で行うことが望ましい。

参考資料

- owasp-mastg Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3) External User Agent vs. Embedded User Agent

ルールブック

- *OAuth2* 認証で使用される主な推奨ライブラリ（推奨）
- *OAuth2* 認証で外部ユーザエージェントを使用する場合はメリット、デメリットを理解して使用する（推奨）

4.3.6 その他の OAuth 2.0 のベストプラクティス

その他のベストプラクティスや詳細情報については、以下のソースドキュメントを参照する。

- 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)

参考資料

- owasp-mastg Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3) Other OAuth2 Best Practices

4.3.7 ルールブック

1. 使用している *JWT* ライブラリの特定及び既知の脆弱性があるかどうかを確認する（必須）
2. トークンは、*KeyStore (Android)* により、モバイルデバイス上に安全に保管されていることを確認する（必須）
3. 期待するハッシュアルゴリズムを要求する必要がある（必須）
4. リフレッシュトークンの有効期限に適切な期間を設定する（必須）
5. *OAuth 2.0* のベストプラクティスの遵守（必須）
6. *OAuth2* 認証で使用される主な推奨ライブラリ（推奨）
7. *OAuth2* 認証で外部ユーザエージェントを使用する場合はメリット、デメリットを理解して使用する（推奨）

4.3.7.1 使用している JWT ライブラリの特定及び既知の脆弱性があるかどうかを確認する（必須）

以下の内容を確認して、脆弱性があるか確認する必要がある。

- 使用している JWT ライブラリを特定し、既知の脆弱性があるかどうかを確認する。
- トークンを含むすべての受信要求について、HMAC が検証されることを確認する。
- 秘密署名鍵または HMAC 秘密鍵の場所を確認する。キーはサーバに保管し、決してクライアントと共有してはならない。発行者と検証者のみが利用可能であるべきである。
- JWT に個人を特定できる情報のような機密データが埋め込まれていないことを確認する。何らかの理由でトークン内にそのような情報を送信する必要があるアーキテクチャの場合、ペイロードの暗号化が適用されていることを確認する。OWASP JWT Cheat Sheet にある Java 実装のサンプルを参照する。
- JWT に一意な識別子を与える jti (JWT ID) クレームでリプレイ攻撃に対処していることを確認する。
- クロスサービスリレー攻撃には、トークンがどのアプリケーションのためのものかを定義する aud (audience) クレームで対処していることを確認する。
- トークンは、KeyStore (Android) により、モバイルデバイス上に安全に保管されていることを確認する。

以下サンプルコードは、KeyStore の暗号化キーにより暗号化したトークンを SharedPreferences へ保存、及び SharedPreferences から取り出して復号処理の一例。

```
val PROVIDER = "AndroidKeyStore"
val ALGORITHM = "RSA"
val CIPHER_TRANSFORMATION = "RSA/ECB/PKCS1Padding"

/**
 * トークンを暗号化して SharedPreferences へ保存する
 */
fun saveToken(context: Context, token: String) {
    val encryptedToken = encrypt(context, "token", token)
```

(次のページに続く)

(前のページからの続き)

```
    val editor = getSharedPreferences(context).edit()
    editor.putString("encrypted_token", encryptedToken).commit()
}

/**
 * トークンを復号して SharedPreferences から取得する
 */
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
}

/**
 * テキストを暗号化する
 * @param context
 * @param alias 対称鍵を識別するためのエリアス。用途ごとに一意にする。
 * @param plainText 暗号化したいテキスト
 * @return 暗号化され Base64 でラップされた文字列
 */
fun encrypt(context: Context, alias: String, plainText: String): String {
    val keyStore = KeyStore.getInstance(PROVIDER)
    keyStore.load(null)

    // 対称鍵がない場合生成
    if (!keyStore.containsAlias(alias)) {
        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)

    // 公開鍵で暗号化
    val cipher = Cipher.getInstance(CIPHER_TRANSFORMATION)
    cipher.init(Cipher.ENCRYPT_MODE, publicKey)
    val bytes = cipher.doFinal(plainText.toByteArray(Charsets.UTF_8))

    // SharedPreferences に保存しやすいように Base64 で String 化
    return Base64.encodeToString(bytes, Base64.DEFAULT)
}

/**
 * 暗号化されたテキストを復化する
 * @param alias 対称鍵を識別するためのエリアス。用途ごとに一意にする。
 * @param encryptedText encrypt で暗号化されたテキスト
 * @return 復号された文字列
```

(次のページに続く)

(前のページからの続き)

```
 */
fun decrypt(alias: String, encryptedText: String): String? {
    val keyStore = KeyStore.getInstance(PROVIDER)
    keyStore.load(null)
    if (!keyStore.containsAlias(alias)) {
        return null
    }

    // 秘密鍵で復号
    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)
}

/**
 * 対称鍵を生成する
 */
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()
}
```

これに違反する場合、以下の可能性がある。

- 使用する JWT ライブラリに関する脆弱性を悪用される。

4.3.7.2 トークンは、KeyStore (Android) により、モバイルデバイス上に安全に保管されていることを確認する（必須）

トークンは KeyStore によりモバイルデバイス上に安全に保管する必要がある。これを怠った場合、第三者によるトークンを利用したサーバ認証が可能になる恐れがある。

KeyStore の使用方法は以下ルールを参照する。

データストレージとプライバシー要件ルールブック

- キーがセキュリティハードウェアの内部に保存されているかどうかを確認する（推奨）

これに違反する場合、以下の可能性がある。

- 第三者によるトークンを利用したサーバ認証が可能になる。

4.3.7.3 期待するハッシュアルゴリズムを要求する必要がある（必須）

攻撃者はトークンを変更し、「 none 」キーワードを使って署名アルゴリズムを変更し、トークンの完全性がすでに検証されていることを示す場合がある。よって、Java アプリケーションでは検証コンテキストを作成する際に、期待するアルゴリズムを明示的に要求する必要がある。

```
// 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);
```

これに違反する場合、以下の可能性がある。

- 「 none 」アルゴリズムで署名されたトークンを、検証済みの署名を持つ有効なトークンであるかのように扱い、アプリケーションは変更されたトークンの要求を信頼する可能性がある。

4.3.7.4 リフレッシュトークンの有効期限に適切な期間を設定する（必須）

リフレッシュトークンの有効期限が適切な期間であることを確認する必要がある。リフレッシュトークンの有効期限内の場合は、新しいアクセストークンの発行をし、有効期限外の場合は、アクセスを拒否してユーザに再ログインを促す必要性がある。

これに違反する場合、以下の可能性がある。

- 有効期限が不適切な場合はログインユーザを第三者が利用できる可能性がある。

4.3.7.5 OAuth 2.0 のベストプラクティスの遵守（必須）

以下の OAuth 2.0 のベストプラクティスを遵守すること。

ユーザエージェント

- ユーザは信頼を視覚的に確認する方法（例：TLS(Transport Layer Security) 確認、ウェブサイトの仕組み）を持つべきである。
- 中間者攻撃を防ぐために、クライアントはサーバの完全修飾ドメイン名を、接続確立時にサーバが提示した公開鍵で検証する必要がある。

付与の種類

- ネイティブアプリでは、暗黙的な付与ではなく、コード付与を使用する必要がある。
- コード付与を使用する場合、コード付与を保護するために PKCE (Proof Key for Code Exchange) を実装する必要があります。サーバ側でも実装されていることを確認する。
- 認可「コード」は有効期限が短く、受信後すぐに使用されるべきである。認可コードが一時的なメモリ上にのみ存在し、保存やログに記録されないことを確認する。

クライアントシークレット

- クライアントはなりすまされる可能性があるため、クライアントの ID を証明するために共有シークレットを使うべきではない（「 client_id 」がすでに証明の役目を果たしている）。もし、クライアントシークレットを使用する場合は、安全なローカルストレージに保存されていることを確認する。

エンドユーザの認証情報

- TLS などのトランスポート層方式でエンドユーザ認証情報の送信を保護する。

トークン

- アクセストークンは一時的なメモリに保存する。
- アクセストークンは、暗号化された接続で送信する必要がある。
- エンドツーエンドの機密性が保証されない場合や、トークンが機密情報や取引へのアクセスを提供する場合は、アクセストークンの範囲と期間を縮小する。
- アプリケーションがアクセストークンをベアラートークンとして使用し、クライアントを識別する他の方法がない場合、トークンを盗んだ攻撃者はそのスコープとそれに関連するすべてのリソースにアクセスできることに留意する。
- リフレッシュトークンは、安全なローカルストレージに保管する。

これに違反する場合、以下の可能性がある。

- 脆弱な OAuth 2.0 の実装となる可能性がある。

4.3.7.6 OAuth2 認証で使用される主な推奨ライブラリ（推奨）

OAuth2 認証の実装を独自開発しないことが重要となる。
OAuth2 認証を実現するために利用されるライブラリは主に以下である。

- Google Play Services (Google)

以下サンプルコードは、Google Play Services での OAuth2 認証の一例。

```
/**
 * Google SignIn
 */
private fun requestOAuth() {
    val signInOptions = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_
↪SIGN_IN).build()
```

(次のページに続く)

(前のページからの続き)

```
    activityResultLauncher.launch(GoogleSignIn.getClient(requireContext(),
↪signInOptions).signInIntent)
}
```

これに注意しない場合、以下の可能性がある。

- 脆弱な OAuth 2.0 の実装となる可能性がある。

4.3.7.7 OAuth2 認証で外部ユーザエージェントを使用する場合はメリット、デメリットを理解して使用する（推奨）

OAuth2 認証は、外部のユーザエージェント（ Chrome や Safari など）を介して行うことできる。外部ユーザエージェントを利用時のメリット、デメリットは以下である。

メリット

- ユーザの認証情報は、決してアプリに直接公開されることはない。このため、ログイン時にアプリが認証情報を取得することはできない（「認証情報のフィッシング」）。
- アプリ自体に認証ロジックをほとんど追加する必要がないため、コーディングミスを防ぐことができる。

デメリット

- ブラウザの動作を制御する方法（証明書のピン留めを有効にする等）がない。

これに注意しない場合、以下の可能性がある。

- 証明書のピン留めを有効にする必要がある際に、利用することができない。

4.4 MSTG-AUTH-4

ユーザがログアウトする際に、リモートエンドポイントは既存のセッションを終了している。

4.4.1 リモートセッション情報の破棄

このテストケースの目的は、ログアウト機能を検証し、それがクライアントとサーバの両方で効果的にセッションを終了させ、ステートレストークンを無効にするかどうかを判断することである。

サーバ側セッションの破棄に失敗することは、ログアウト機能の実装で最もよくあるエラーの一つである。このエラーは、ユーザがアプリケーションからログアウトした後も、セッションやトークンを保持し続ける。有効な認証情報を取得した攻撃者は、その情報を使い続け、ユーザのアカウントを乗っ取ることが可能である。

多くのモバイルアプリは、ユーザを自動的にログアウトさせない。顧客にとって不便だから、あるいはステートレス認証の実装時に決定されたからなど、さまざまな理由が考えられる。しかし、アプリケーションにはログアウト機能が必要であり、ベストプラクティスに従って実装し、ローカルに保存されたトークンやセッション識別子をすべて破棄する必要がある。セッション情報がサーバに保存されている場合、そのサーバにログア

ウト要求を送信することによっても破棄されなければならない。リスクの高いアプリケーションの場合、トークンを無効にする必要がある。トークンやセッション識別子を破棄しないと、トークンが流出した場合にアプリケーションに不正にアクセスされる可能性がある。なお、適切に消去されない情報は、デバイスのバックアップ時など、後で流出する可能性があるため、他の機密情報も同様に破棄する必要がある。

4.4.2 静的解析

サーバコードが利用可能な場合は、ログアウト機能によってセッションが正しく終了することを確認する。この検証は、テクノロジーによって異なる。以下は、サーバ側のログアウトが正しく行われるためのセッション終了の様々な例である。

- Spring (Java)
- Ruby on Rails
- PHP

ステートレス認証でアクセストークンやリフレッシュトークンを使用する場合は、モバイルデバイスから破棄する必要がある。リフレッシュトークンはサーバ上で無効化する必要がある。

4.4.3 動的解析

動的アプリケーション解析用のインターセプションプロキシを使用し、以下の手順を実行して、ログアウトが正しく実装されているかどうかを確認する。

1. アプリケーションにログインする。
2. 認証を必要とするリソースにアクセスする。通常は、アカウントに属する個人情報の要求である。
3. アプリケーションからログアウトする。
4. 手順 2 の要求を再送信して、再度データにアクセスする。

サーバ上でログアウトが正しく実装されている場合は、エラーメッセージまたはログインページへのリダイレクトがクライアントに返される。一方、手順 2 で得たのと同じレスポンスを受け取る場合、トークンあるいはセッション ID はまだ有効で、サーバ上で正しく終了していない。OWASP ウェブテストガイド (WSTG-SESS-06 には、詳しい説明とより多くのテストケースが含まれている。

参考資料

- owasp-mastg Testing User Logout (MSTG-AUTH-4)

ルールブック

- **アプリにログイン機能が存在する場合のリモートセッション情報の破棄のベストプラクティス（推奨）**

4.4.4 ルールブック

1. アプリにログイン機能が存在する場合のリモートセッション情報の破棄のベストプラクティス（推奨）

4.4.4.1 アプリにログイン機能が存在する場合のリモートセッション情報の破棄のベストプラクティス（推奨）

アプリにログイン機能が存在する場合、以下のベストプラクティスに従いリモートセッション情報を破棄する。

- アプリケーションにログアウト機能を用意する。
- ログアウト時にローカルに保存されたトークンやセッション識別子をすべて破棄する。
- セッション情報がサーバに保存されている場合、サーバにログアウト要求を送信して破棄する。
- リスクの高いアプリの場合、トークンを無効にする。

トークンやセッション識別子を破棄しないと、トークンが流出した場合にアプリケーションに不正にアクセスされる可能性がある。破棄されない情報は、デバイスのバックアップ時など、後で流出する可能性があるため、他の機密情報も同様に破棄する必要がある。

これに注意しない場合、以下の可能性がある。

- ログイン情報がメモリ内に残り、第三者に利用される。

4.5 MSTG-AUTH-5

パスワードポリシーが存在し、リモートエンドポイントで実施されている。

4.5.1 パスワードポリシー遵守

パスワードの強度は、認証にパスワードが使用される場合の重要な懸念事項である。パスワードポリシーは、エンドユーザが遵守すべき要件を定義するものである。パスワードポリシーは通常、パスワードの長さ、パスワードの複雑さ、およびパスワードのトポロジーを指定する。「強力な」パスワードポリシーは、手動または自動によるパスワードクラッキングを困難または不可能にする。以下のセクションでは、パスワードのベストプラクティスに関するさまざまな領域をカバーする。より詳細な情報は、OWASP Authentication Cheat Sheet を参照する。

参考資料

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6)

4.5.2 静的解析

パスワードポリシーの存在を確認し、長さと無制限の文字セットに焦点を当てた OWASP Authentication Cheat Sheet に従って、実装されたパスワードの複雑さの要件を検証すること。ソースコードに含まれるパスワード関連の関数をすべて特定し、それぞれで検証チェックが行われていることを確認する。パスワード検証機能を確認し、パスワードポリシーに違反するパスワードが拒否されることを確認する。

参考資料

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) Static Analysis

ルールブック

- 「強力な」パスワードポリシー（推奨）

4.5.2.1 zxcvbn

zxcvbn は、パスワードクラッカーにヒントを得て、パスワードの強度を推定するために使用できる一般的なライブラリである。JavaScript で利用可能であるが、サーバ側では他の多くのプログラミング言語でも利用可能である。インストール方法は様々なので、Github のリポジトリを確認する。インストールすると、zxcvbn を使って、パスワードの複雑さと解読に必要な推測回数を計算することができる。

HTML ページに zxcvbn JavaScript ライブラリを追加した後、ブラウザのコンソールで zxcvbn コマンドを実行すると、スコアを含むパスワードのクラックの可能性に関する詳細情報を取得できる。

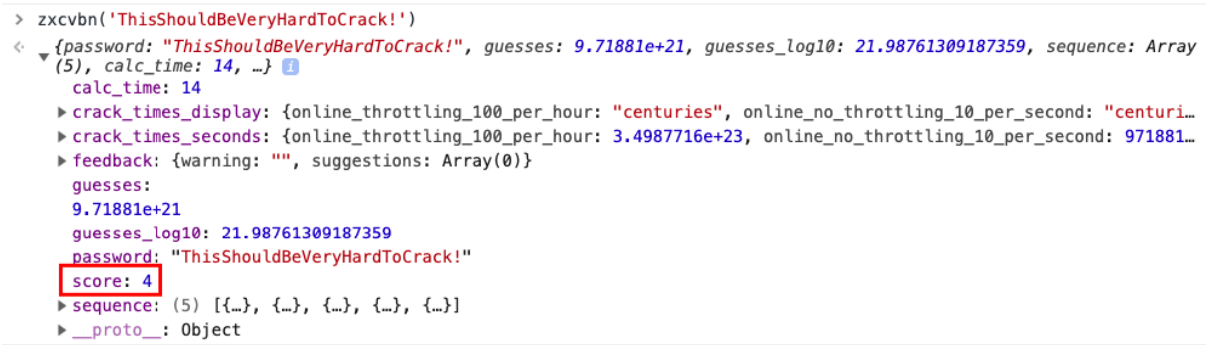


図 4.5.2.1.1　zxcvbn コマンド例

スコアは以下のように定義され、例えばパスワードの強度バーに使用することができる。

```
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)
```

(次のページに続く)

(前のページからの続き)

```
4 # very unguessable: strong protection from offline slow-hash scenario. (guesses >
↪= 10^10)
```

なお、zxcvbn はアプリ開発者が Java（または他の）実装を使って実装することもでき、ユーザに強力なパスワードを作成するように導くことができる。

参考資料

- 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

一要素認証スキーム（パスワードのみなど）に対する辞書攻撃の成功の可能性をさらに低くするために、パスワードがデータ漏洩で流出したかどうかを検証することができる。これは、Troy Hunt 氏による Pwned Passwords API をベースにしたサービス（ api.pwnedpasswords.com で入手可能）を使って行うことができる。例えば、コンパニオンウェブサイトの「Have I been pwned ? 」である。この API は、パスワード候補の SHA-1 ハッシュをもとに、指定されたパスワードのハッシュが、このサービスが収集したさまざまなデータ漏洩の事例の中で見つかった回数を返す。ワークフローは以下のステップを実行する。

- ユーザ入力を UTF-8 にエンコードする。(例：the password test)
- ステップ 1 の結果の SHA-1 ハッシュを取得する。(例：test のハッシュは A94A8FE5CC...)
- 最初の 5 文字（ハッシュプレフィックス）をコピーし、次の API を用いて範囲検索に使用する：http GET https://api.pwnedpasswords.com/range/A94A8
- 結果を繰り返し、ハッシュの残りを探す。(たとえば、FE5CC... が返されたリストの一部であるかどうか）もしそれが返されたリストの一部でなければ、与えられたハッシュのパスワードは見つからない。そうでなければ、FE5CC... のように、それが違反で見つかった回数を示すカウンターを返す。(例：FE5CC... ：76479)

Pwned Passwords API に関する詳細なドキュメントは、オンラインで確認できる。

なお、この API は、ユーザが登録やパスワードの入力を行う際に、推奨パスワードかどうかを確認するために、アプリ開発者が使用するのが最適である。

参考資料

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) Have I Been Pwned: PwnedPasswords

ルールブック

- **推奨パスワードかの確認（推奨）**

4.5.3.1 ログイン制限

スロットリングの手順をソースコードで確認する：

指定されたユーザ名で短時間にログインを試行した場合のカウンターと、最大試行回数に達した後にログインを試行しない方法を提供する。認証されたログイン試行後は、エラーカウンターをリセットすること。

ブルートフォース対策として、以下のベストプラクティスを遵守する。

- ログインに数回失敗したら、対象のアカウントをロックし（一時的または恒久的に）、それ以降のログイン試行を拒否すべき。
- 一時的なアカウントロックには、5 分間のアカウントロックが一般的に使用される。
- クライアント側の制御は簡単にバイパスされるため、制御はサーバ上で行う必要がある。
- 不正なログインの試行は、特定のセッションではなく、対象となるアカウントについて集計する必要がある。

その他のブルートフォース軽減技術については、OWASP のブルートフォース攻撃の阻止のページに記載されている。

参考資料

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) Have I Been Pwned: PwnedPasswords Login Throttling

ルールブック

- **ブルートフォース対策として、以下のベストプラクティスを遵守する（必須）**

4.5.4 ルールブック

1. **「強力な」パスワードポリシー（推奨）**
2. **推奨パスワードかの確認（推奨）**
3. **ブルートフォース対策として、以下のベストプラクティスを遵守する（必須）**

4.5.4.1 「強力な」パスワードポリシー（推奨）

以下は「強力な」パスワードを設定するためのポリシーの一例である。

- パスワードの長さ
  - 最小：8 文字未満のパスワードは脆弱であると見なされる。
  - 最大：一般的な最大長は 64 文字である。long password Denial of Service attacks による攻撃を防ぐために最大長を設定することは必要である。
- パスワードをユーザに通知なしで切り捨てない。

- パスワードの構成規則 Unicode と空白を含むすべての文字の使用を許可する。
- 資格情報のローテーションパスワードの漏洩が発生、または特定されたときに資格情報のローテーションを行う必要がある。
- パスワード強度メーターユーザが複雑なパスワードを作成する、過去に特定されたパスワードの設定をブロックするためにパスワード強度メーターを用意する。

以下サンプルコードは、Android アプリで対応すべきパスワードの長さ判定と、パスワードの構成規則の処理の一例。

レイアウトでの例:

```
<EditText
    android:id="@+id/passwordText"
    android:hint="password"
    android:maxLength="64"/>
```

Kotlin での例:

```
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)
}
```

参考資料

- OWASP CheatSheetSeries Implement Proper Password Strength Controls

これに注意しない場合、以下の可能性がある。

- 脆弱なパスワードとなり予測される。

4.5.4.2 推奨パスワードかの確認（推奨）

Pwned Passwords API を使用することで、ユーザが登録やパスワードの入力を行う際に、推奨パスワードかどうかを確認することができる。

これに注意しない場合、以下の可能性がある。

- 脆弱なパスワードとなり予測される。

4.5.4.3 ブルートフォース対策として、以下のベストプラクティスを遵守する（必須）

ブルートフォース対策として、以下のベストプラクティスを遵守する。

- ログインに数回失敗したら、対象のアカウントをロックし（一時的または恒久的に）、それ以降のログイン試行を拒否すべき。
- 一時的なアカウントロックには、5 分間のアカウントロックが一般的に使用される。
- クライアント側の制御は簡単にバイパスされるため、制御はサーバ上で行う必要がある。
- 不正なログインの試行は、特定のセッションではなく、対象となるアカウントについて集計する必要がある。

その他のブルートフォース軽減技術については、OWASP のブルートフォース攻撃の阻止のページに記載されている。

※サーバ側のルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- ブルートフォース攻撃に脆弱になる。

4.6 MSTG-AUTH-6

リモートエンドポイントは過度な資格情報の送信に対する保護を実装している。

※リモートサービス側での対応に関する章であるため、本資料ではガイド記載を省略

参考資料

- owasp-mastg Testing Best Practices for Passwords Dynamic Testing(MSTG-AUTH-6)

4.7 MSTG-AUTH-7

事前に定義された非アクティブ期間およびアクセストークンの有効期限が切れた後に、セッションはリモートエンドポイントで無効にしている。

※リモートサービス側での対応に関する章であるため、本資料ではガイド記載を省略

参考資料

- owasp-mastg Testing Session Timeout(MSTG-AUTH-7)

4.8 MSTG-AUTH-12

認可モデルはリモートエンドポイントで定義および実施されている。

※リモートサービス側での対応に関する章であるため、本資料ではガイド記載を省略

5 ネットワーク通信要件

5.1 MSTG-NETWORK-1

データはネットワーク上で TLS を使用して暗号化されている。セキュアチャネルがアプリ全体を通して一貫して使用されている。

5.1.1 安全なネットワークリクエスト

5.1.1.1 推奨されるネットワーク API の使い方

まず、ソースコード内ですべてのネットワークリクエストを特定し、平文の HTTP URL が使用されていないことを確認する必要がある。機密情報は、HttpsURLConnection または SSLSocket (TLS を使用したソケットレベルの通信用) を使用して、安全なチャネルで送信されるようにする。

次に、セキュアな接続を行うことを前提とした低レベルの API (SSLSocket など) を使用する場合でも、セキュアな実装が必要であることに注意する。例えば、SSLSocket はホスト名を検証しない。ホスト名を確認するには getDefaultHostnameVerifier を使用する。コード例は Android の開発者向けドキュメントを参照する。

参考資料

- 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

ルールブック

- **平文の *HTTP URL* が使用されていないことを確認する（必須）**
- **機密情報は安全なチャネルで送信されるようにする（必須）**
- **低レベルの *API* を使用したセキュアな実装（必須）**

5.1.1.2 平文の HTTP トラフィックの設定

次に、アプリが平文の HTTP トラフィックを許可していないことを確認する必要がある。Android 9 (API level 28) 以降、平文の HTTP トラフィックはデフォルトでブロックされる（ デフォルトのネットワークセキュリティ構成によって ）が、アプリが平文を送信する方法はまだ複数ある。

- AndroidManifest.xml ファイルの <application> タグの android:usesCleartextTraffic 属性を設定する。Network Security Configuration が設定されている場合、このフラグは無視されることに注意する。
- <domain-config> 要素で cleartextTrafficPermitted 属性を true に設定し、CleartextTraffic を有効にするように Network Security Configuration を設定する。
- 低レベルの API (例：Socket) を使用して、カスタム HTTP 接続を設定する。
- クロスプラットフォームフレームワーク (Flutter、Xamarin など) を使用する。これらには通常、HTTP ライブラリの独自の実装がある。

上記のすべてのケースは、全体として注意深く分析する必要がある。例えば、アプリが Android Manifest や Network Security Configuration で CleartextTraffic を許可していない場合でも、実際には HTTP トラフィックを送信している可能性がある。これは、低レベルの API (Network Security Configuration が無視される) を使用している場合や、クロスプラットフォームフレームワークが適切に設定されていない場合に起こり得る。

詳細については、「HTTPS と SSL によるセキュリティ」を参照。

参考資料

- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1) Testing for Cleartext Traffic

ルールブック

- **アプリが平文の *HTTP* トラフィックを許可していないことを確認する（必須）**

5.1.2 ルールブック

1. **平文の *HTTP URL* が使用されていないことを確認する（必須）**
2. **機密情報は安全なチャネルで送信されるようにする（必須）**
3. **低レベルの *API* を使用したセキュアな実装（必須）**
4. **アプリが平文の *HTTP* トラフィックを許可していないことを確認する（必須）**

5.1.2.1 平文の HTTP URL が使用されていないことを確認する（必須）

ソースコード内ですべてのネットワークリクエストを特定し、平文の HTTP URL が使用されていないことを確認する必要がある。

これに違反する場合、以下の可能性がある。

- 平文情報が第三者に漏洩する。

5.1.2.2 機密情報は安全なチャネルで送信されるようにする（必須）

危険なチャネル（ HTTP ）により機密情報を送信すると、平文のまま送信されることにより第三者へ漏洩する可能性がある。そのため、機密情報を送信する場合は安全なチャネル（ HTTPS、SSL 等）で送信する必要がある。

以下に安全なチャネルで送信するためのサンプルコードを示す。

- 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
}
```

これに違反する場合、以下の可能性がある。

- 機密情報が第三者に漏洩する。

5.1.2.3 低レベルの API を使用したセキュアな実装（必須）

低レベルの API を使用する場合でも、セキュアな実装が必要である。SSLSocket はホスト名を検証しない。ホスト名を確認するには getDefaultHostnameVerifier を使用する。

以下は SSLSocket 使用時のホスト名検証サンプルコードの一例。

```
    // 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)) {
```

(次のページに続く)

(前のページからの続き)

```
            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()
```

これに違反する場合、以下の可能性がある。

- 通信先ホストが信頼できるか保証されない可能性がある。

5.1.2.4 アプリが平文の HTTP トラフィックを許可していないことを確認する（必須）

アプリが平文の HTTP トラフィックを許可していないことを確認する。Android 9 (API level 28) 以降、平文の HTTP トラフィックはデフォルトでブロックされるが、アプリが平文を送信する方法は複数存在する。

以下はアプリから平文を送信する方法の一例。

- AndroidManifest.xml ファイルの <application> タグの android:usesCleartextTraffic 属性を設定する。Network Security Configuration が設定されている場合、このフラグは無視されることに注意する。

```
<application
            android:usesCleartextTraffic="true">
</application>
```

- <domain-config> 要素で cleartextTrafficPermitted 属性を true に設定し、CleartextTraffic を有効にするように Network Security Configuration を設定する。

```
<?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>
```

- 低レベルの API (例：Socket) を使用して、カスタム HTTP 接続を設定する。

```
val address = InetSocketAddress(ip, port)
val socket = Socket()
try {
    socket.connect(address)
} catch (e: Exception) {
}
```

- クロスプラットフォームフレームワーク (Flutter、Xamarin など) を使用する。これらには通常、HTTP ライブラリの独自の実装がある。

これに違反する場合、以下の可能性がある。

- HTTP トラフィックにより平文を送信する。

5.2 MSTG-NETWORK-2

TLS 設定は現在のベストプラクティスと一致している。モバイルオペレーティングシステムが推奨される標準規格をサポートしていない場合には可能な限り近い状態である。

5.2.1 推奨される TLS 設定

サーバ側で適切な TLS 設定を行うことも重要である。SSL プロトコルは非推奨であり、もはや使用するべきではない。また、TLS v1.0 と TLS v1.1 には既知の脆弱性があり、2020 年までにすべての主要なブラウザでその使用が非推奨となった。TLS v1.2 および TLS v1.3 は、安全なデータ通信のためのベストプラクティスと考えられている。

Android 10(API level 29) 以降では、TLS v1.3 がデフォルトで有効になり、より高速で安全な通信が可能になる。TLS v1.3 の主な変更点は、暗号スイートのカスタマイズができなくなり、TLS v1.3 を有効にするとすべての暗号スイートが有効になるのに対し、0-RTT(Zero Round Trip) モードがサポートされない。

クライアントとサーバの両方が同じ組織で管理され、互いに通信するためだけに使用されている場合、設定を強化することでセキュリティを強化できる。

モバイルアプリケーションが特定のサーバに接続する場合、そのネットワークスタックを調整することで、サーバの構成に対して可能な限り高いセキュリティレベルを確保することができる。オペレーティングシステムのサポートが不十分な場合、モバイルアプリケーションはより弱い構成を使用せざるを得なくなる可能性がある。

参考資料

- owasp-mastg Verifying the TLS Settings (MSTG-NETWORK-2) Recommended TLS Settings

ルールブック

- **安全な通信プロトコル（必須）**

5.2.2 推奨される暗号スイート

暗号スイートの構造は以下の通りである。

```
Protocol_KeyExchangeAlgorithm_WITH_BlockCipher_IntegrityCheckAlgorithm
```

この構造には以下が含まれる。

- 暗号化で使用されるプロトコル
- TLS ハンドシェイク中にサーバとクライアントが認証に使用する鍵交換アルゴリズム
- メッセージストリームの暗号化に使用されるブロック暗号
- メッセージの認証に使用される完全性保証チェックアルゴリズム

例：TLS_RSA_WITH_3DES_EDE_CBC_SHA

上記の例では、以下の暗号化スイートが使用されている。

- プロトコルとしての TLS
- 認証のための RSA 非対称暗号化
- EDE_CBC モードによる対称暗号化のための 3DES
- 完全性のための SHA ハッシュアルゴリズム

TLSv1.3 では、鍵交換アルゴリズムは暗号スイートの一部ではなく、TLS ハンドシェイク中に決定されることに注意する。

次のリストでは、暗号スイートの各部分のさまざまなアルゴリズムを紹介する。

プロトコル :

- SSL v1
- SSL v2 - RFC 6176
- SSL v3 - RFC 6101
- TLS v1.0 - RFC 2246
- TLS v1.1 - RFC 4346
- TLS v1.2 - RFC 5246
- TLS v1.3 - RFC 8446

鍵交換アルゴリズム :

- 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

ブロック暗号 :

- 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

完全性チェックアルゴリズム :

- MD5 - RFC 6151
- SHA - RFC 6234
- SHA256 - RFC 6234
- SHA384 - RFC 6234

暗号スイートの効率は、そのアルゴリズムの効率に依存することに注意する必要がある

以下のリソースは、TLS で使用するために推奨される最新の暗号スイートが含まれている。

- IANA が推奨する暗号スイートは、TLS Cipher Suites に記載されている。
- OWASP が推奨する暗号スイートは、TLS Cipher String Cheat Sheet に記載されている。

Android の一部バージョンでは、推奨する暗号スイートに対応していないものもあるため、互換性のために、Android のバージョンでサポートされている暗号スイートを確認し、上位の暗号スイートを選択することが可能である。

サーバが適切な暗号スイートをサポートしているかどうかを確認する場合は、さまざまなツールを使用できる。

- testssl.sh は、「TLS/SSL 暗号、プロトコル、およびいくつかの暗号の欠陥のサポートについて、任意のポートでサーバのサービスをチェックする無料のコマンドラインツール」である。

最後に、HTTPS 接続が終了するサーバまたは終了プロキシが、ベストプラクティスに従って設定されていることを確認する。OWASP Transport Layer Protection cheat sheet および Qualys SSL/TLS Deployment Best Practices を参照する。

参考資料

- owasp-mastg Verifying the TLS Settings (MSTG-NETWORK-2) Cipher Suites Terminology

ルールブック

- *TLS* で推奨される暗号化スイート（推奨）

5.2.3 ルールブック

1. 安全な通信プロトコル（必須）
2. *TLS* で推奨される暗号化スイート（推奨）

5.2.3.1 安全な通信プロトコル（必須）

サーバ側で適切な TLS 設定を行うことも重要である。SSL プロトコルは非推奨であり、もはや使用するべきではない。

非推奨プロトコル

- SSL
- TLS v1.0
- TLS v1.1

TLS v1.0 と TLS v1.1 については、2020 年までにすべての主要なブラウザでその使用が非推奨となった。

推奨プロトコル

- TLS v1.2
- TLS v1.3

Android 10(API level 29) 以降では、TLS v1.3 がデフォルトで有効になり、より高速で安全な通信が可能になる。TLS v1.3 を有効にするとすべての暗号スイートが有効になるのに対し、0-RTT(Zero Round Trip) モードがサポートされない。

これに違反する場合、以下の可能性がある。

- セキュリティエクスプロイトに対して脆弱である。

5.2.3.2 TLS で推奨される暗号化スイート（推奨）

以下は推奨される暗号化スイートの一例。(TLS Cipher Suites で推奨されている暗号化スイートの中で、Android で非推奨ではないものを記載。)

- 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

これに注意しない場合、以下の可能性がある。

- 脆弱な暗号化スイートを使用する可能性がある。

5.3 MSTG-NETWORK-3

セキュアチャネルが確立されたときに、アプリはリモートエンドポイントの X.509 証明書を検証している。信頼された CA により署名された証明書のみが受け入れられている。

5.3.1 信頼する証明書の設定

5.3.1.1 ターゲット SDK バージョンごとのデフォルト設定

Android 7.0 (API level 24) 以降をターゲットとするアプリケーションは、ユーザが提供する CA を信頼しないデフォルトのネットワークセキュリティ構成を使用し、ユーザを誘い込んで悪意のある CA をインストールさせる MITM 攻撃の可能性を低減する。

apktool を使用してアプリをデコードし、apktool.yml の targetSdkVersion が 24 以降であることを確認する。

```
grep targetSdkVersion UnCrackable-Level3/apktool.yml
  targetSdkVersion: '28'
```

ただし、targetSdkVersion >=24 の場合でも、開発者はカスタムネットワークセキュリティ構成を使用してデフォルトの保護を無効にし、ユーザが提供する CA をアプリに強制的に信頼させる custom trust anchor を定義することができる。「**カスタムトラストアンカーの分析**」を参照。

参考資料

- owasp-mastg Testing Endpoint Identify Verification (MSTG-NETWORK-3) Static Analysis Verifying the Target SDK Version

ルールブック

- **ターゲット *SDK* バージョンによる *MITM* 攻撃の可能性（必須）**
- **カスタムトラストアンカーの分析（必須）**

5.3.1.2 カスタムトラストアンカーの分析

Network Security Configuration ファイルを検索し、<certificates src="user"> を定義しているカスタムの <trust-anchors> を調査する (これは避けるべきである) 。

エントリの優先順位を慎重に分析する必要がある。

- <domain-config> のエントリまたは親の <domain-config> に値が設定されていない場合、設定は <base-config> に基づいて行われる。
- このエントリで定義されていない場合、デフォルトの設定が使用される。

Android 9 (API level 28) を対象とするアプリのネットワークセキュリティ構成の例は以下の通りである。

```
<?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>
```

いくつかの考察を紹介する。:

- <base-config> がないため、Android 9 (API level 28) 以降のデフォルト設定が他のすべての接続に使用される（原則としてシステム認証 CA のみが信頼される)。
- しかし、<domain-config> はデフォルトの設定を上書きし、指定された <domain> (owasp.org) に対して、アプリがシステムとユーザの両方の認証局を信頼することを可能にする。
- includeSubdomains="false" のため、サブドメインには影響しない。

すべてをまとめると、上記のネットワークセキュリティ構成を次のように説明することができる。「このアプリは、サブドメインを除く owasp.org ドメインのシステムとユーザの両方の認証局を信頼する。他のドメインでは、アプリはシステムの認証局のみを信頼する。」

参考資料

- owasp-mastg Testing Endpoint Identify VerificatioTG-NETWORK-3) Static Analysis Analyzing Custom Trust Anchorsn (MS

ルールブック

- **カスタムトラストアンカーの分析（必須）**

5.3.2 サーバ証明書の検証

5.3.2.1 TrustManager による検証

TrustManager は、Android において信頼できる接続を確立するために必要な条件を確認する手段である。このとき、以下の条件を確認する必要がある。

- 証明書は、信頼できる CA によって署名されているか？
- 証明書の有効期限が切れていないか？
- 自己署名の証明書であるか？

以下のコード スニペットは開発中に使用されることがあり、関数 checkClientTrusted, checkServerTrusted, getAcceptedIssuers をオーバーライドして、どんな証明書でも受け入れてしまう。このような実装は避けるべきであり、必要な場合は、組み込みのセキュリティ上の欠陥を回避するために、production builds とは明確に分離する必要がある。

```
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());
```

参考資料

- owasp-mastg Testing Endpoint Identify Verification (MSTG-NETWORK-3) Static Analysis Verifying the Server Certificate

ルールブック

- *TrustManager* による検証（必須）

5.3.2.2 WebView でのサーバ証明書の検証

アプリケーションは WebView を使用して、アプリケーションに関連付けられた Web サイトをレンダリングすることがある。これは、アプリケーションのインタラクションに内部 WebView を使用する Apache Cordova などの HTML/JavaScript ベースのフレームワークに当てはまる。WebView が使用される場合、モバイルブラウザはサーバ証明書の検証を実行する。WebView がリモート Web サイトに接続しようとしたときに発生する TLS エラーを無視することは、バッドプラクティスである。

以下のコードは、WebView に提供される WebViewClient のカスタム実装と全く同じように、TLS エラーを無視する。

```
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();
    }
});
```

Apache Cordova フレームワークの内部 WebView 使用の実装は、application manifest で android:debuggable フラグが有効になっていると、onReceivedSslError メソッドで TLS エラーを無視する。そのため、アプリがデバッグ可能でないことを確認する。

参考資料

- 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

ルールブック

- *WebView* でのサーバ証明書の検証のバッドプラクティス（必須）

5.3.3 ホスト名の検証

クライアント側の TLS 実装におけるもう 1 つのセキュリティ上の欠陥は、ホスト名の検証の欠如である。開発環境は通常、有効なドメイン名ではなく内部アドレスを使用するため、開発者はホスト名の検証を無効にし（あるいはアプリケーションに任意のホスト名を許可させ）、アプリケーションが本番稼働するときに変更することを忘れてしまうことがある。次のコードは、ホスト名検証を無効にするものである。

```
final static HostnameVerifier NO_VERIFY = new HostnameVerifier() {
    public boolean verify(String hostname, SSLSession session) {
        return true;
```

(次のページに続く)

(前のページからの続き)

```
    }
};
```

組み込みの HostnameVerifier を使用すると、任意のホスト名を受け入れることができる。

```
HostnameVerifier NO_VERIFY = org.apache.http.conn.ssl.SSLSocketFactory
                             .ALLOW_ALL_HOSTNAME_VERIFIER;
```

信頼できる接続を設定する前に、アプリケーションがホスト名を検証していることを確認する。

参考資料

- owasp-mastg Testing Endpoint Identify Verification (MSTG-NETWORK-3) Static Analysis Hostname Verification

ルールブック

- ホスト名の検証（必須）

5.3.4 ルールブック

1. ターゲット *SDK* バージョンによる *MITM* 攻撃の可能性（必須）
2. カスタムトラストアンカーの分析（必須）
3. *TrustManager* による検証（必須）
4. *WebView* でのサーバ証明書の検証のバッドプラクティス（必須）
5. ホスト名の検証（必須）

5.3.4.1 ターゲット SDK バージョンによる MITM 攻撃の可能性（必須）

Android 7.0 (API level 24) 以降をターゲットとするアプリケーションは、ユーザが提供する CA を信頼しないデフォルトのネットワークセキュリティ構成を使用し、ユーザを誘い込んで悪意のある CA をインストールさせる MITM 攻撃の可能性を低減する。

apktool を使用してアプリをデコードし、apktool.yml の targetSdkVersion が 24 以降であることを確認する。

これに違反する場合、以下の可能性がある。

- 悪意のある CA をインストールさせる MITM 攻撃の可能性が高まる。

5.3.4.2 カスタムトラストアンカーの分析（必須）

targetSdkVersion >=24 の場合でも、開発者はカスタムネットワークセキュリティ構成を使用してデフォルトの保護を無効にし、ユーザが提供する CA をアプリに強制的に信頼させることをカスタムトラストアンカーで定義することができる。

AndroidManifest.xml に設定されている android:networkSecurityConfig の設定を確認する必要がある。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application android:networkSecurityConfig="@xml/network_security_config"
                    ... >
        ...
    </application>
</manifest>
```

android:networkSecurityConfig に設定されている Network Security Configuration ファイルを確認して、以下のタグの状態を確認する必要がある。

- <base-config>
- <trust-anchors>
- <certificates>

※ <certificates src="user"> の設定は避ける必要がある。

また、固有の構成で設定されていないタグは <base-config> での設定を継承し、<base-config> が設定されていない場合はプラットフォームの既定値が設定される。

エントリの優先順位を慎重に分析する必要がある。

- <domain-config> のエントリまたは親の <domain-config> に値が設定されていない場合、設定は <base-config> に基づいて行われる。
- このエントリで定義されていない場合、デフォルトの設定が使用される。

Android 9 (API level 28) を対象とするアプリのネットワークセキュリティ構成の例は以下の通りである。

```
<?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>
```

いくつかの考察を紹介する。:

- <base-config> がないため、Android 9 (API level 28) 以降のデフォルト設定が他のすべての接続に使用される（原則としてシステム認証 CA のみが信頼される）。
- しかし、<domain-config> はデフォルトの設定を上書きし、指定された <domain> (owasp.org) に対して、アプリがシステムとユーザの両方の認証局を信頼することを可能にする。
- includeSubdomains="false" のため、サブドメインには影響しない。

すべてをまとめると、上記のネットワークセキュリティ構成を次のように説明することができる。「このアプリは、サブドメインを除く owasp.org ドメインのシステムとユーザの両方の認証局を信頼する。他のドメインでは、アプリはシステムの認証局のみを信頼する。」

これに違反する場合、以下の可能性がある。

- 悪意のある CA をインストールさせる MITM 攻撃の可能性が高まる。

5.3.4.3 TrustManager による検証（必須）

TrustManager を用いて関数 checkClientTrusted, checkServerTrusted, getAcceptedIssuers をオーバーライドした場合、以下サンプルコードのようにクライアント証明書の検証を行わずに全ての証明書を受け入れてしまうと、安全な通信を保証できない。開発時の場合には、以下サンプルコードにより自己証明書での動作確認が実施できて便利であるが、誤って製品版に組み込まれないようにするために処理を分けるべきである。

```
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());
```

以下サンプルコードは、特定の CA のセットを信頼するために、TrustManager を初期化作成して HttpsURLConnection を設定する処理である。

```
    // Load CAs from an InputStream
    // (could be from a resource or ByteArrayInputStream or ...)
```

(次のページに続く)

(前のページからの続き)

```
    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);
```

参考資料

- HTTPS と SSL を使用したセキュリティ 未知の認証局

これに違反する場合、以下の可能性がある。

- 自己証明書での検証が含まれる場合、信頼できる証明書であるかの判断がつかない。

5.3.4.4 WebView でのサーバ証明書の検証のバッドプラクティス（必須）

アプリケーションは WebView を使用して、アプリケーションに関連付けられた Web サイトをレンダリングすることがある。これは、アプリケーションのインタラクションに内部 WebView を使用する Apache Cordova などの HTML/JavaScript ベースのフレームワークに当てはまる。WebView が使用される場合、モバイルブラウザはサーバ証明書の検証を実行する。WebView がリモート Web サイトに接続しようとしたときに発生する TLS エラーを無視することは、バッドプラクティスである。

以下のサンプルコードは、TLS エラーを無視して WebViewClient に Web サイトをロードする処理の一例。

```
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();
    }
});
```

Apache Cordova フレームワークの内部 WebView 使用の実装は、application manifest で android:debuggable フラグが有効になっていると、onReceivedSslError メソッドで TLS エラーを無視する。そのため、アプリがデバッグ可能でないことを確認する。

これに違反する場合、以下の可能性がある。

- 中間者攻撃に対して脆弱になる。

5.3.4.5 ホスト名の検証（必須）

開発段階において、開発者はホスト名の検証を無効（あるいはアプリケーションに任意のホスト名を許可させ）にして開発を行っていることがある。本番環境稼働時に変更せず検証を無効としていることがある。

以下は、無効としている場合のものである。

```
final static HostnameVerifier NO_VERIFY = new HostnameVerifier() {
    public boolean verify(String hostname, SSLSession session) {
        return true;
    }
};
```

以下は、任意のホスト名を受け入れるようにしたものである。

```
HostnameVerifier NO_VERIFY = org.apache.http.conn.ssl.SSLSocketFactory
                             .ALLOW_ALL_HOSTNAME_VERIFIER;
```

本番環境接続時にホスト名の検証を行う必要がある。

これに違反する場合、以下の可能性がある。

- ホスト先が信頼される宛先のホストでは無い状態で通信する可能性がある。

6 プラットフォーム連携要件

6.1 MSTG-PLATFORM-1

アプリは必要となる最低限のパーミッションのみを要求している。

6.1.1 パーミッションの保護レベルの種類

Android では、インストールされたすべてのアプリに個別のシステム ID (Linux のユーザ ID とグループ ID) が割り当てられる。Android の各アプリはプロセスサンドボックスで動作するため、アプリはサンドボックスの外にあるリソースやデータへのアクセスを明示的に要求する必要がある。アプリは、システムのデータや機能を使用するために必要なパーミッションを宣言することで、このアクセスを要求する。データや機能の機密性や重要性に応じて、Android システムは自動的にアクセス許可を与えるか、ユーザにリクエストの承認を求める。

Android のパーミッションは、提供される保護レベルに基づいて 4 つの異なるカテゴリに分類される。

- Nomal : このパーミッションは、他のアプリ、ユーザ、およびシステムに対するリスクを最小限に抑えながら、アプリケーションレベルの分離された機能へのアクセスをアプリに許可する。Android 6.0 (API level 23) 以降を対象とするアプリの場合、これらのパーミッションはインストール時に自動的に付与される。それより前の API level を対象とするアプリの場合、インストール時にユーザが承認する必要がある。例: android.permission.INTERNET.
- Dangerous : このパーミッションは通常、アプリにユーザデータの制御や、ユーザに影響を与えるような方法でのデバイスの制御を与える。このタイプのパーミッションは、インストール時に付与されない場合がある。アプリがこのパーミッションを持つべきかどうかは、ユーザの判断に委ねられる場合がある。例: android.permission.RECORD_AUDIO.
- Signature : このパーミッションは、要求元のアプリが、パーミッションを宣言したアプリの署名に使用されたものと同じ証明書で署名されている場合にのみ付与される。このパーミッションは、インストール時に付与される。例: android.permission.ACCESS_MOCK_LOCATION.
- SystemOrSignature : このパーミッションは、システムイメージに組み込まれたアプリケーション、またはパーミッションを宣言したアプリケーションの署名に使用されたのと同じ証明書で署名されたアプリ

ケーションにのみ付与される。例: android.permission.ACCESS_DOWNLOAD_MANAGER.

すべてのパーミッションの一覧は、Android 開発者向けドキュメントに記載されており、その具体的な手順も記載されている。

- アプリのマニフェストファイルでアプリのパーミッションを宣言する。
- プログラムによってアプリのパーミッションを要求する。
- カスタムアプリパーミッションを定義して、アプリのリソースと機能を他のアプリと共有する。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Overview

6.1.2 API level ごとのパーミッションに関する変更点

6.1.2.1 Android 8.0 (API level 26) の変更点

以下の変更は、Android 8.0 (API level 26) 上で動作するすべてのアプリに影響し、より低い API level を対象とするアプリにも適用される。

- 連絡先プロバイダーの使用状況統計の変更 : アプリが READ_CONTACTS パーミッションを要求した場合、連絡先の使用状況データのクエリは正確な値ではなく、近似値を返す (オートコンプリート API はこの変更の影響を受けない) 。

Android 8.0 (API level 26) 以降を対象とするアプリは、以下の影響を受ける。

- アカウントへのアクセスおよび検出性の向上 : GET_ACCOUNTS パーミッションを付与されたアプリは、Authenticator がアカウントを所有しているか、ユーザがそのアクセスを許可していない限り、ユーザアカウントにアクセスすることができなくなった。
- 新しいテレフォニーパーミッション : 以下のパーミッション (dangerous に分類される) が、PHONE パーミッショングループに含まれるようになった。
  - ANSWER_PHONE_CALLS パーミッションにより、(acceptRingingCall を介して) プログラムで着信電話に応答できる。
  - READ_PHONE_NUMBERS パーミッションは、デバイスに保存されている電話番号への読み取りアクセスを許可する。
- dangerous パーミッションを付与する場合の制限 : dangerous パーミッションはパーミッショングループに分類される (例 : STORAGE グループには READ_EXTERNAL_STORAGE と WRITE_EXTERNAL_STORAGE が含まれる) 。Android 8.0 (API level 26) 以前は、グループの 1 つのパーミッションを要求するだけで、そのグループのすべてのパーミッションも同時に許可された。これは、Android 8.0 (API level 26) から変更された。アプリが実行時にパーミッションを要求すると、システムはその特定のパーミッションのみを許可する。ただし、そのパーミッショングループのパーミッションに対するそれ以降のリクエストは、ユーザにパーミッションのダイアログを表示することなく、自動的に許可されることに注意する。Android 開発者向けドキュメントにある以下の例を参照してください。

あるアプリが、マニフェストに READ_EXTERNAL_STORAGE と WRITE_EXTERNAL_STORAGE の両方を記載しているとする。アプリは READ_EXTERNAL_STORAGE を要求し、ユーザはそれを許可する。アプリのターゲットが API level 25 以下の場合、システムは WRITE_EXTERNAL_STORAGE も同時に許可する。これは、同じ STORAGE 権限グループに属し、マニフェストにも登録されているためである。アプリが Android 8.0 (API level 26) をターゲットにしている場合、システムはその時点で READ_EXTERNAL_STORAGE のみを許可する。しかし、アプリが後で WRITE_EXTERNAL_STORAGE を要求すると、システムはユーザに促さずに直ちにその権限を許可する。

パーミッショングループのリストは、Android 開発者向けドキュメントで確認することができる。また、もう少し分かりやすくするために、Google は、特定のパーミッションが Android SDK の将来のバージョンで 1 つのグループから別のグループに移動する可能性があるため、アプリのロジックはこれらのパーミッショングループの構造に依存すべきではないと警告している。ベストプラクティスは、必要なときにすべてのパーミッションを明示的に要求することである。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Android 8.0 (API level 26) Changes

ルールブック

- **必要なときにすべてのパーミッションを明示的に要求する（必須）**

6.1.2.2 Android 9 (API level 28) の変更点

以下の変更は、Android 9 上で動作するすべてのアプリに影響し、API level 28 未満を対象とするアプリにも影響する。

- 通話履歴へのアクセス制限： EAD_CALL_LOG 、 WRITE_CALL_LOG 、および PROCESS_OUTGOING_CALLS (dangerous) パーミッションは、PHONE から新しい CALL_LOG パーミッショングループに移動した。これは、電話をかけることができる (例: PHONE グループのパーミッションを付与されている) だけでは、通話履歴へのアクセスを得るには十分でないことを意味する。
- 電話番号へのアクセス制限 : Android 9 (API level 28) で動作する場合、電話番号を読み取りたいアプリは READ_CALL_LOG パーミッションが必要である。
- Wi-Fi の位置情報、接続情報へのアクセス制限 : SSID と BSSID の値を取得することはできない (例：WifiManager.getConnectionInfo 経由で以下のすべてが true でない限り)
  - ACCESS_FINE_LOCATION または ACCESS_COARSE_LOCATION パーミッション。
  - ACCESS_WIFI_STATE パーミッション。
  - 位置情報サービスが有効になっていること (「設定」→「位置情報」) 。

Android 9 (API level 28) 以降を対象とするアプリは、以下の影響を受ける。

- Build serial number deprecation : デバイスのハードウェアシリアル番号は、READ_PHONE_STATE (dangerous) パーミッションが付与されていない限り、(Build.getSerial などを介して) 読み取ることはできない。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Android 9 (API Level 28) Changes

ルールブック

- 必要なときにすべてのパーミッションを明示的に要求する（必須）

6.1.2.3 Android 10 (API level 29) の変更点

Android 10 (API level 29) では、ユーザのプライバシーを強化する機能がいくつか導入されている。パーミッションに関する変更は、より低い API level を対象とするものを含め、Android 10 (API level 29) 上で動作するすべてのアプリケーションに影響する。

- 位置情報アクセスの制限：位置情報アクセスに「アプリ使用時のみ」の許可オプションを追加した。
- デフォルトでスコープされたストレージ：Android 10 (API level 29) をターゲットとするアプリは、外部ストレージのアプリ固有のディレクトリにあるファイルにアクセスするためのストレージパーミッションを宣言する必要はなく、メディアストアから作成されたファイルにもアクセスできる。
- 画面コンテンツへのアクセス制限：READ_FRAME_BUFFER 、CAPTURE_VIDEO_OUTPUT 、および CAPTURE_SECURE_VIDEO_OUTPUT パーミッションが署名アクセスのみになり、デバイスの画面コンテンツへのサイレントアクセスが防止される。
- レガシーアプリのユーザ向けパーミッションチェック：Android 5.1 (API level 22) 以前をターゲットとするアプリを初めて実行する場合、ユーザは特定のレガシーパーミッション (以前はインストール時に自動的に付与されていたもの) へのアクセスを取り消すことができるパーミッション画面が表示されるようになる。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Android 10 (API level 29) Changes

ルールブック

- 必要なときにすべてのパーミッションを明示的に要求する（必須）

6.1.3 他のアプリコンポーネントとの連携

6.1.3.1 他のアプリの Activity との連携

パーミッションは、マニフェストの タグ内にある android:permission 属性を介して適用される。これらのパーミッションは、どのアプリケーションがその Activity を開始できるかを制限する。パーミッションは Context.startActivity と Activity.startActivityForResult の間にチェックされる。必要なパーミッションを保持していない場合、呼び出しから SecurityException が throw される。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Activity Permission Enforcement

6.1.3.2 他のアプリの Service との連携

マニフェストの <service> タグ内にある android:permission 属性によって適用されるパーミッションは、関連付けられた Service を開始またはバインドできるユーザを制限する。パーミッションは、Context.startService 、Context.stopService 、および Context.bindService で確認される。必要なパーミッションを保持していない場合、呼び出しから SecurityException が throw される。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Service Permission Enforcement

6.1.3.3 他のアプリの BroadcastReceiver との連携

<receiver> タグ内の android:permission 属性によって適用されるパーミッションは、関連付けられた BroadcastReceiver に対してブロードキャストを送信するためのアクセスを制限するものである。パーミッションは、Context.sendBroadcast が返された後、送信されたブロードキャストを与えられたレシーバーに配信しようとする際に確認される。必要なパーミッションを保持していなくても例外は throw されず、結果は未送信のブロードキャストとなる。

Context.registerReceiver にパーミッションを与えることで、プログラムで登録した receiver に対して誰がブロードキャストできるかを制御することができる。逆に、Context.sendBroadcast を呼び出す際にパーミッションを与えることで、どの BroadcastReceiver がブロードキャストを受信することができるかを制限することができる。

receiver と broadcaster の両方がパーミッションを必要とすることがあることに注意する。この場合、関連するターゲットに Intent を送信するためには、両方のパーミッションのチェックをパスする必要がある。詳細については、Android 開発者向けドキュメントの「権限の設定によるブロードキャストの制限」セクションを参照する。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Broadcast Permission Enforcement

ルールブック

- *Broadcast* の受信側と送信側の両方で権限を必要とする場合があることに注意する（必須）

6.1.3.4 他のアプリの ContentProvider との連携

<provider> タグ内の android:permission 属性によって適用されるパーミッションは、ContentProvider 内のデータへのアクセスを制限する。ContentProvider には、次に説明する URI パーミッションと呼ばれる重要な追加のセキュリティ機能がある。ContentProvider は他のコンポーネントとは異なり設定可能な 2 つの別々の許可属性を持っている。android:readPermission は provider から読み取ることができるユーザを制限し、android:writePermission は provider に書き込むことができるユーザを制限する。ContentProvider が読み取りと書き込みの両方のパーミッション保護されている場合、書き込みパーミッションのみを保持しても読み取りパーミッションは付与されない。

パーミッションは、最初に provider を取得するときと、ContentProvider を使用して操作を実行するときにチェックされる。ContentResolver.query を使用する場合は、読み取りパーミッションを保持する必要があり、

ContentResolver.insert,ContentResolver.update.ContentResolver.delete を使用する場合は書き込みパーミッションが必要である。これらすべてのケースで適切なパーミッションが保持されていない場合、呼び出しから SecurityException が throw される。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Content Provider Permission Enforcement

6.1.4 ContentProvider の URI パーミッション

ContentProvider を使用する場合、標準の許可システムでは十分ではない。例えば、ContentProvider は、情報を取得するためにカスタム URI を使用しながら、自分自身を保護するために、パーミッションを READ パーミッションに制限したい場合がある。アプリケーションは、その特定の URI に対するパーミッションのみを持つべきである。

解決策は、URI ごとのパーミッションである。activity を開始するとき、または結果を返すとき、メソッドは Intent.FLAG_GRANT_READ_URI_PERMISSION と Intent.FLAG_GRANT_WRITE_URI_PERMISSION の両方またはどちらか一方を設定することができる。これにより、ContentProvider からのデータにアクセスするパーミッションがあるかどうかに関係なく、特定の URI の activity にパーミッションが付与される。

これにより、ユーザの操作によってきめ細かいパーミッションをアドホックに付与する、一般的な capability-style モデルが可能になる。これは、アプリが必要とするパーミッションを、アプリの動作に直接関連するものだけに減らすための重要な機能となりえる。このモデルがないと、悪意のあるユーザが他のメンバの電子メールの添付ファイルにアクセスしたり、保護されていない URI を介して将来使用するために連絡先リストを取得したりする可能性がある。マニフェストでは、android:grantUriPermissions 属性またはノードが URI を制限するのに役立つ。

URI パーミッションのドキュメント

- grantUriPermission
- revokeUriPermission
- checkUriPermission

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Content Provider URI Permissions
- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Documentation for URI Permissions

ルールブック

- **アプリケーションは特定の *URI* に対するパーミッションのみを持つべき（必須）**

6.1.5 カスタムパーミッション

Android では、アプリが Service やコンポーネントを他のアプリに公開することができる。公開されたコンポーネントにアプリがアクセスするためには、カスタムパーミッションが必要である。android:name と android:protectionLevel の 2 つの必須属性を持つパーミッションタグを作成することで、AndroidManifest.xml でカスタムパーミッションを定義できる。

カスタムパーミッションは、「最小権限の原則」に従って作成することが重要である。パーミッションは、その目的に応じて、意味のある正確なラベルと説明で明示的に定義する必要がある。

以下は、START_MAIN_ACTIVITY というカスタムパーミッションの例で、TEST_ACTIVITY Activity を起動する際に必要なパーミッションである。

最初のコードブロックは新しいパーミッションを定義しており、これは自明である。label タグはパーミッションの概要で、description はその概要のより詳細なバージョンである。付与されるパーミッションの種類に応じて、protectionLevel を設定することができる。パーミッションを定義したら、アプリのマニフェストに追加することでそれを強制することができる。この例では、2 番目のブロックが、作成したパーミッションで制限するコンポーネントを表している。これは android:permission 属性を追加することで強制することができる。

```
<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>
```

START_MAIN_ACTIVITY が作成されると、アプリは AndroidManifest.xml ファイル内の uses-permission タグを使用してそれを要求できる。START_MAIN_ACTIVITY が付与されたアプリは、TEST_ACTIVITY を起動することができる。<uses-permission android:name="myapp.permission.START_MAIN_ACTIVITY"> は、<application> の前に宣言する必要があることに注意する。そうしないと、実行時に exception が発生する。パーミッションの概要と manifest-intro に基づいている以下の例を参照する。

```
<manifest>
<uses-permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY" />
        <application>
            <activity>
            </activity>
        </application>
</manifest>
```

他のアプリとの衝突を避けるため、上記の例 (com.domain.application.permission) のようにパーミッションを登録する際には、reverse-domain annotation を使用することを推奨する。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Custom Permissions

ルールブック

- **カスタムパーミッションは、「最小権限の原則」に従って作成する（必須）**
- **カスタムパーミッションは *reverse-domain annotation* で登録する（推奨）**

6.1.6 静的解析

6.1.6.1 Android のパーミッション

アプリに本当に必要なパーミッションかどうかを確認し、不要なパーミッションを削除する。例えば、AndroidManifest.xml ファイルの INTERNET パーミッションは、Activity が Web ページを WebView に読み込むために必要である。ユーザは、dangerous パーミッションを使用するアプリケーションの権利を取り消すことができるため、開発者は、そのパーミッションを必要とするアクションが実行されるたびに、アプリケーションが適切なパーミッションを持っているかどうかをチェックする必要がある。

```
<uses-permission android:name="android.permission.INTERNET" />
```

開発者と一緒にパーミッションを確認し、すべてのパーミッションの目的を特定し、不要なパーミッションを削除する。

AndroidManifest.xml ファイルを手動で調べる以外に、Android Asset Packaging ツール (aapt) を使って APK ファイルのパーミッションを調べることも可能である。

aapt は Android SDK の build-tools フォルダに付属している。入力として APK ファイルを必要としている。インストールされているアプリの一覧表示にあるように、adb shell pm list packages -f | grep -i <keyword> を実行することで、デバイス内の APK を一覧表示することができる。

```
$ 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'
```

また、adb や dumpsys ツールでより詳細なパーミッションのリストを取得することも可能である。

```
$ 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:
```

dangerous とされる一覧のパーミッションの説明については、このパーミッションの概要を参照する。

```
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
```

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Android Permissions

ルールブック

- **アプリに本当に必要なパーミッションかどうかを確認し、不要なパーミッションを削除する（必須）**

6.1.6.2 カスタムパーミッション

アプリケーションマニフェストファイルによるカスタムパーミッションの強制とは別に、プログラムによるパーミッションのチェックも可能である。しかし、この方法はエラーが発生しやすく、runtime instrumentation などで簡単にバイパスできるため、推奨されない。activity が指定されたパーミッションを持っているかどうかをチェックするには、ContextCompat.checkSelfPermission メソッドを呼び出すことを推奨する。次のスニペットのようなコードが表示されたら、マニフェストファイルで同じパーミッションが強制されていることを確認する。

```
private static final String TAG = "LOG";
int canProcess = checkCallingOrSelfPermission("com.example.perm.READ_INCOMING_MSG
↪");
if (canProcess != PERMISSION_GRANTED)
throw new SecurityException();
```

または、マニフェストファイルと比較する ContextCompat.checkSelfPermission を使用する。

```
if (ContextCompat.checkSelfPermission(secureActivity.this, Manifest.READ_INCOMING_
↪MSG)
        != PackageManager.PERMISSION_GRANTED) {
            //!= stands for not equals PERMISSION_GRANTED
            Log.v(TAG, "Permission denied");
        }
```

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Android Permissions

ルールブック

- **プログラムによるカスタムパーミッションのチェック（推奨）**

6.1.7 パーミッションのリクエスト

アプリケーションに実行時に要求する必要のあるパーミッションがある場合、アプリケーションはそれを取得するために requestPermissions メソッドを呼び出す必要がある。アプリは必要なパーミッションと指定した integer 型の request code を非同期にユーザに渡し、ユーザが同じスレッドでリクエストの受け入れまたは拒否を選択すると返す。レスポンスが返された後、同じ request code がアプリのコールバックメソッドに渡される。

```
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.");
}
```

一度呼び出されたシステムダイアログボックスは変更できないため、ユーザに何らかの情報や説明を提供する必要がある場合は、requestPermissions の呼び出しの前に行う必要があることに注意する。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Requesting Permissions

ルールブック

- **必要なときにすべてのパーミッションを明示的に要求する（必須）**

6.1.8 パーミッションリクエストの応答処理

ここで、アプリはシステムメソッド onRequestPermissionsResult をオーバーライドして、パーミッションが付与されたかどうかを確認する必要がある。このメソッドは、入力パラメータとして integer 型の requestCode を受け取る (requestPermissions で作成されたものと同じ requestCode である)。

WRITE_EXTERNAL_STORAGE には、次のコールバックメソッドを使用できる。

```
@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.
    }
}
```

パーミッションは、同じグループからの同様のパーミッションがすでにリクエストされている場合でも、必要なパーミッションごとに明示的にリクエストする必要がある。Android 7.1 (API level 25) 以前のアプリケーションでは、ユーザがあるパーミッショングループの要求されたパーミッションの 1 つを許可すると、Android はそのグループのすべてのパーミッションを自動的にアプリケーションに付与する。Android 8.0 (API level 26) 以降では、ユーザが同じ許可グループの許可をすでに与えている場合でも、許可は自動的に与えられますが、アプリケーションは許可を明示的に要求する必要がある。この場合、ユーザの操作なしに onRequestPermissionsResult ハンドラが自動的にトリガーされる。

例えば、READ_EXTERNAL_STORAGE と WRITE_EXTERNAL_STORAGE の両方が Android Manifest にリストされているが、READ_EXTERNAL_STORAGE にのみパーミッションが付与されている場合、

WRITE_EXTERNAL_STORAGE をリクエストすると、同じグループであり明示的にリクエストされていないためユーザの介入なしに自動的にパーミッションを持つことができるようになる。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Handling Responses to Permission Requests

ルールブック

- **必要なときにすべてのパーミッションを明示的に要求する（必須）**

6.1.9 パーミッション分析

アプリケーションが実際に必要なパーミッションを要求しているかどうか、常に確認する。特に DANGEROUS と SIGNATURE パーミッションは、扱いを誤るとユーザとアプリケーションの両方に影響を与える可能性があるため、アプリの目的に関係のないパーミッションが要求されていないことを確認する。例えば、シングルプレイのゲームアプリが android.permission.WRITE_SMS へのアクセスを要求していたら、怪しいと思うはずである。

パーミッションを分析する際には、アプリの具体的なユースケースシナリオを調査し、使用中の DANGEROUS なパーミッションに代わる API があるかどうかを常にチェックする必要がある。その良い例が、SMS ベースのユーザ認証を行う際に SMS パーミッションの使用を効率化する SMS Retriever API である。この API を使用することで、アプリケーションは DANGEROUS パーミッションを宣言する必要がなくなり、ユーザとアプリケーションの開発者の両方にとって、Permissions Declaration Form を提出する必要がなくなるという利点がある。

参考資料

- owasp mastg Testing App Permissions (MSTG-PLATFORM-1) Permission Analysis

ルールブック

- **アプリに本当に必要なパーミッションかどうかを確認し、不要なパーミッションを削除する（必須）**

6.1.10 動的解析

インストールされたアプリケーションのパーミッションは adb で取得することができる。以下の抜粋は、アプリケーションで使用されているパーミッションを調べる方法を示している。

```
$ 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
```

(次のページに続く)

(前のページからの続き)

```
  android.permission.USE_CREDENTIALS: granted=true
  com.google.android.providers.gsf.permission.READ_GSERVICES: granted=true
...
```

出力は、以下のカテゴリを使用してすべてのパーミッションを表示する。

- declared permissions: すべてのカスタムパーミッションのリスト
- requested and install permissions: normal および signature パーミッションを含むすべてのインストール時パーミッションのリスト
- runtime permissions: すべての dangerous なパーミッションのリスト

動的解析を行う場合:

- 要求されたパーミッションがアプリに本当に必要かどうかを評価する。例えば、android.permission.WRITE_SMS へのアクセスを必要とするシングルプレイヤーゲームは、良いアイデアではないかもしれない。
- 多くの場合、アプリはパーミッションを宣言する代わりに、次のような方法を選択することができる。
  - ACCESS_FINE_LOCATION の代わりに ACCESS_COARSE_LOCATION パーミッションを要求する。あるいは、パーミッションをまったく要求せず、代わりに郵便番号を入力するようユーザに要求することもできる。
  - CAMERA パーミッションを要求する代わりに、ACTION_IMAGE_CAPTURE または ACTION_VIDEO_CAPTURE intent 動作を呼び出す。
  - Bluetooth デバイスとのペアリング時に、ACCESS_FINE_LOCATION 、ACCESS_COARSE_LOCATIION 、BLUETOOTH_ADMIN パーミッションを宣言する代わりに、Companion Device Pairing (Android 8.0 (API level 26) 以降) を使用する。
- Privacy Dashboard (Android 12 (API level 31) 以降) を使用して、アプリが機密情報へのアクセスについてどのように説明しているかを確認する。

特定のパーミッションについて詳細を知りたい場合は、Android のドキュメントを参照する。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Dynamic Analysis

ルールブック

- **アプリに本当に必要なパーミッションかどうかを確認し、不要なパーミッションを削除する（必須）**

6.1.11 ルールブック

1. 必要なときにすべてのパーミッションを明示的に要求する（必須）
2. *Broadcast* の受信側と送信側の両方で権限を必要とする場合があることに注意する（必須）
3. アプリケーションは特定の *URI* に対するパーミッションのみを持つべき（必須）
4. カスタムパーミッションは、「最小権限の原則」に従って作成する（必須）
5. カスタムパーミッションは *reverse-domain annotation* で登録する（推奨）
6. アプリに本当に必要なパーミッションかどうかを確認し、不要なパーミッションを削除する（必須）
7. プログラムによるカスタムパーミッションのチェック（推奨）

6.1.11.1 必要なときにすべてのパーミッションを明示的に要求する（必須）

Google は、特定のパーミッションが Android SDK の将来のバージョンで 1 つのグループから別のグループに移動する可能性があるため、アプリのロジックはこれらのパーミッショングループの構造に依存すべきではないと警告している。したがって、ベストプラクティスは、必要なときにすべてのパーミッションを明示的に要求することである。

以下は明示的に外部ストレージへの書き込み権限取得のサンプルコードを示す。

AndroidManifest.xml へ外部ストレージへの書き込み権限を宣言。

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

必要なタイミングで権限を要求するサンプルコード。

```
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);
```

(次のページに続く)

(前のページからの続き)

```
        // 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.");
}
```

権限要求の応答のサンプルコード。

```
@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.
    }
}
```

これに違反する場合、以下の可能性がある。

- 特定のパーミッションが Android SDK の将来のバージョンで 1 つのグループから別のグループに移動した場合にアプリロジックの見直しが必要となる。

6.1.11.2 Broadcast の受信側と送信側の両方で権限を必要とする場合があることに注意する（必須）

権限を設定して、Broadcast の対象を一定の権限を持つアプリセットに制限できる。Broadcast の送信側または受信側に対して制限を適用できる。その場合、受信側と送信側の両方で権限を必要とすることに注意する。

権限を設定した送信

sendBroadcast(Intent, String) または sendOrderedBroadcast(Intent, String, BroadcastReceiver, Handler, int, String, Bundle) を呼び出す際に、権限パラメータを指定できる。マニフェストでタグを使用して権限をリクエストしたレシーバー（また、安全性の理由により合わせて権限を付与されたレシーバー）のみが、Broadcast を受信できる。

以下のサンプルコードは 権限パラメータを指定 して Broadcast を送信する一例。

sendBroadcast

```
sendBroadcast(Intent("com.example.NOTIFY"), Manifest.permission.SEND_SMS)
```

sendOrderedBroadcast

Broadcast を受信するには、受信側のアプリが以下のサンプルコードのように権限をリクエストする必要がある。

```
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);
```

SEND_SMS などの既存のシステム権限を指定することも、<permission> 要素を使用してカスタム権限を定義することもできる。

権限を設定した受信

BroadcastReceiver の登録時に権限パラメータを指定する場合（registerReceiver(BroadcastReceiver, IntentFilter, String, Handler) または Manifest の <receiver> タグを使用）は、Manifest で <uses-permission> タグを使用して権限をリクエストした Broadcast の送信側（また、安全性の理由により合わせて権限を付与された送信側）のみが、intent を Receiver に送信できる。

以下に示すように、受信側のアプリに Manifest で宣言された Receiver があるとする。

```
<receiver android:name=".MyBroadcastReceiver"
              android:permission="android.permission.SEND_SMS">
        <intent-filter>
            <action android:name="android.intent.action.AIRPLANE_MODE"/>
        </intent-filter>
    </receiver>
```

または、受信側のアプリにコンテキスト登録された Receiver があるとする。

```
var filter = IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED)
registerReceiver(receiver, filter, Manifest.permission.SEND_SMS, null )
```

上記の Receiver に Broadcast を送信できるようにするには、以下のサンプルコードのように、送信側のアプリから権限をリクエストする必要がある。

```
<uses-permission android:name="android.permission.SEND_SMS"/>
```

これに違反する場合、以下の可能性がある。

- Broadcast の対象を制限することができない。

6.1.11.3 アプリケーションは特定の URI に対するパーミッションのみを持つべき（必須）

ContentProvider は、情報を取得するためにカスタム URI を使用しながら、自分自身を保護するために、パーミッションを READ パーミッションに制限したい場合がある。アプリケーションは、その特定の URI に対するパーミッションのみを持つべきである。

解決策は、URI ごとのパーミッションである。activity を開始するとき、または結果を返すとき、メソッドは Intent.FLAG_GRANT_READ_URI_PERMISSION と Intent.FLAG_GRANT_WRITE_URI_PERMISSION の両方またはどちらか一方を設定することができる。これにより、ContentProvider からのデータにアクセスするパーミッションがあるかどうかに関係なく、特定の URI の activity にパーミッションが付与される。

以下に URI ごとのパーミッション 指定し付与する例を示す。

以下サンプルコードは、パーミッション許可用の AndroidManifest.xml での宣言。

```
<provider android:name=".MyProvider"
          android:authorities="com.example.sampleprovider.myprovider"
          android:grantUriPermissions="true" />
```

以下サンプルコードは、パーミッションの許可と、許可終了用の処理例。

```
// grantUriPermission により許可
grantUriPermission("com.example.sampleresolver",
                   Uri.parse("content://com.example.sampleprovider/myprovider/"),
                   Intent.FLAG_GRANT_READ_URI_PERMISSION);

//以降アクセスした場合は読み込み (query) のみ許可される。

// revokeUriPermission により許可を止める
revokeUriPermission(Uri.parse("content://com.example.sampleprovider/myprovider/"),
                    Intent.FLAG_GRANT_READ_URI_PERMISSION);
```

これにより、ユーザの操作によってきめ細かいパーミッションをアドホックに付与する、一般的な capability-style モデルが可能になる。これは、アプリが必要とするパーミッションを、アプリの動作に直接関連するものだけに減らすための重要な機能となりえる。このモデルがないと、悪意のあるユーザが他のメンバの電子メールの添付ファイルにアクセスしたり、保護されていない URI を介して将来使用するために連絡先リストを取得したりする可能性がある。

これに違反する場合、以下の可能性がある。

- 意図しないアプリからの ContentProvider へのアクセスを制限できない。

6.1.11.4 カスタムパーミッションは、「最小権限の原則」に従って作成する（必須）

カスタムパーミッションは、「最小権限の原則」に従って作成することが重要である。パーミッションは、その目的に応じて、意味のある正確なラベルと説明で明示的に定義する必要がある。

以下にカスタムパーミッションを明示的に指定し、他アプリで使用する例を示す。

以下は、「START_MAIN_ACTIVITY」というカスタムパーミッションの宣言例。

```
<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>
```

以下は、作成された「START_MAIN_ACTIVITY」を他アプリで使用するための宣言。

```
<manifest>
<uses-permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY" />
        <application>
            <activity>
            </activity>
        </application>
</manifest>
```

これに違反する場合、以下の可能性がある。

- 異常が発生した場合に意図しないセキュリティ侵害を受ける可能性がある。

6.1.11.5 カスタムパーミッションは reverse-domain annotation で登録する（推奨）

カスタムパーミッションは reverse-domain annotation で登録することが推奨される。

以下は、ドメインが「example.com」の場合にカスタムパーミッションを reverse-domain annotation で登録する例。

```
<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" />
```

これに注意しない場合、以下の可能性がある。

- 登録するカスタムパーミッション名が他のアプリと衝突する可能性がある。

6.1.11.6 アプリに本当に必要なパーミッションかどうかを確認し、不要なパーミッションを削除する（必須）

ユーザは、DANGEROUS パーミッションを使用するアプリケーションの権利を取り消すことができるため、開発者は、そのパーミッションを必要とするアクションが実行されるたびに、アプリケーションが適切なパーミッションを持っているかどうかをチェックする必要がある。

開発者と一緒にパーミッションを確認し、すべてのパーミッションの目的を特定し、不要なパーミッションを削除する。

以下は AndroidManifest.xml でのパーミッションの宣言。

```
<uses-permission android:name="android.permission.INTERNET" />
```

パーミッションの分析

また、パーミッションを分析し、アプリの具体的なユースケースシナリオを調査することで、使用中の DANGEROUS なパーミッションに代わる API があるかどうかを常にチェックする必要がある。その良い例が、SMS ベースのユーザ認証を行う際に SMS パーミッションの使用を効率化する SMS Retriever API である。この API を使用することで、アプリケーションは DANGEROUS パーミッションを宣言する必要がなくなり、ユーザとアプリケーションの開発者の両方にとって、Permissions Declaration Form を提出する必要がなくなるという利点がある。

分析方法の一つとして、Privacy Dashboard (Android 12 (API level 31) 以降) を使用し、アプリが機密情報へのアクセスについてどのように説明しているかを確認方法もある。

これに違反する場合、以下の可能性がある。

- 不要な DANGEROUS パーミッションが存在する場合は、ユーザに対する不要なパーミッション要求ロジックが必要となる。

6.1.11.7 プログラムによるカスタムパーミッションのチェック（推奨）

アプリケーションマニフェストファイルによるカスタムパーミッションの強制とは別に、プログラムによるパーミッションのチェックも可能である。

その一つの方法として、ContextCompat.checkSelfPermission メソッドによるチェックを推奨する。

以下サンプルコードは、ContextCompat.checkSelfPermission メソッドによるカスタムパーミッションのチェック処理である。

```
if (ContextCompat.checkSelfPermission(secureActivity.this, Manifest.READ_INCOMING_
↪MSG)
        != PackageManager.PERMISSION_GRANTED) {
            //!= stands for not equals PERMISSION_GRANTED
```

(次のページに続く)

(前のページからの続き)

```
            Log.v(TAG, "Permission denied");
        }
```

上記とは別に、checkCallingOrSelfPermission メソッドによりチェックする方法も存在する。こちらの方法はエラーが発生しやすく、runtime instrumentation などで簡単にバイパスできるため、推奨されない。

以下サンプルコードは、checkCallingOrSelfPermission メソッドによるカスタムパーミッションのチェック処理である。

```
private static final String TAG = "LOG";
int canProcess = checkCallingOrSelfPermission("com.example.perm.READ_INCOMING_MSG
↪");
if (canProcess != PERMISSION_GRANTED)
throw new SecurityException();
```

このようなプログラムによるチェック処理を実施する場合は、マニフェストファイルで同じパーミッションが強制されていることを確認する。

これに注意しない場合、以下の可能性がある。

- 不要な DANGEROUS パーミッションが存在する場合は、ユーザに対する不要なパーミッション要求ロジックが必要となる。

6.2 MSTG-PLATFORM-2

外部ソースおよびユーザからの入力はすべて検証されており、必要に応じてサニタイズされている。これには UI、intent やカスタム URL などの IPC メカニズム、ネットワークソースを介して受信したデータを含んでいる。

6.2.1 クロスサイトスクリプティングの問題

クロスサイトスクリプティング（ XSS ）の問題は、攻撃者がユーザが閲覧するウェブページにクライアント側のスクリプトを埋め込むことができるようにするものである。このタイプの脆弱性は、ウェブアプリケーションに広く存在する。ユーザが埋め込まれたスクリプトをブラウザで閲覧すると、攻撃者は同一オリジンポリシーをバイパスする権限を取得し、様々な悪用（セッションクッキーの窃盗、キー押下のログ、任意のアクションの実行など）が可能になる。

ネイティブアプリの場合、Web ブラウザに依存しないため、XSS のリスクははるかに低くなる。しかし、iOS の WKWebView や非推奨の UIWebView 、Android の WebView などの WebView コンポーネントを使用したアプリは、このような攻撃に対して潜在的な脆弱性を持っている。

古い例だが、よく知られているのは、Phil Purviance が最初に発見した、iOS 用 Skype アプリのローカル XSS 問題である。この Skype アプリは、メッセージの送信者名を適切にエンコードしておらず、攻撃者は、ユーザがメッセージを閲覧した際に実行される悪意のある JavaScript を埋め込むことが可能であった。Phil は概念実証の中で、この問題を悪用し、ユーザのアドレス帳を盗む方法を示した。

参考資料

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2)

6.2.1.1 静的解析

WebView を確認して、アプリからレンダリングされた信頼できない入力がないか調査する。

WebView によって開かれた URL の一部がユーザの入力によって決定される場合、XSS 問題が存在する可能性がある。次の例は、Linus Särud 氏によって報告された Zoho Web Service における XSS 問題である。

Java

```
webView.loadUrl("javascript:initialize(" + myNumber + ");");
```

Kotlin

```
webView.loadUrl("javascript:initialize($myNumber);")
```

ユーザ入力によって決まる XSS 問題のもう一つの例は、パブリックオーバーライドされたメソッドである。

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 氏は、以下の HackerOne のレポートで、これを利用することができた。Quora の ActionBarContentActivity では、HTML パラメータへの入力はすべて信頼されることになる。ペイロードは、adb 、ModalContentActivity 経由のクリップボードデータ、およびサードパーティアプリケーションからの Intents を使用して成功した。

- 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>'
```

- クリップボードデータ

```
$ am start -n com.quora.android/com.quora.android.ModalContentActivity  \
-e url 'http://test/test' -e html \
'<script>alert(QuoraAndroid.getClipboardData());</script>'
```

- Java/Kotlin でのサードパーティからの Intent の使用

```
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)
```

WebView を使用してリモート Web サイトを表示する場合、HTML をエスケープする負担はサーバ側に移行する。Web サーバに XSS の欠陥が存在する場合、これを利用して WebView のコンテキストでスクリプトを実行することができる。そのため、Web アプリケーションのソースコードに対して静的解析を行うことが重要である。

以下のベストプラクティスに従っていることを確認する。

- HTML 、JavaScript 、その他の解釈されるコンテキストでは、絶対に必要な場合を除き、信頼できないデータはレンダリングされない。
- エスケープ文字には、HTML エンティティエンコーディングなど、適切なエンコーディングが適用される。注: JavaScript ブロック内にある URL をレンダリングするなど、HTML が他のコード内にネストされている場合、エスケープ ルールは複雑になる。

レスポンスでデータがどのようにレンダリングされるかを検討する。例えば、データが HTML コンテキストでレンダリングされる場合、エスケープする必要がある 6 つの制御文字が存在する。

表 6.2.1.1.1　エスケープする必要がある制御文字一覧

Character	Escaped
&	&amp;
<	&lt;
>	&gt;
"	&quot;
'	&#x27;
/	&#x2F;

エスケープルールやその他の防止策の包括的なリストについては、OWASP XSS Prevention Cheat Sheet を参照

する。

参考資料

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2) Static Analysis

ルールブック

- *WebView* を確認して、アプリからレンダリングされた信頼できない入力がないか調査する（必須）

6.2.1.2 動的解析

HTML タグや特殊文字を利用可能なすべての入力フィールドに注入し、ウェブアプリケーションが無効な入力を拒否するか、出力中の HTML メタ文字をエスケープするかを検証することで、XSS 問題を最もよく検出することができる。

reflected XSS 攻撃とは、悪意のあるリンクを経由して悪意のあるコードが挿入されるエクスプロイトを指す。このような攻撃をテストするためには、自動入力ファジングが有効な方法と考えられている。例えば、BURP Scanner は、reflected XSS 攻撃の脆弱性を特定するのに非常に有効である。自動解析の場合と同様に、テストパラメータを手動で確認し、すべての入力ベクトルがカバーされていることを確認する。

参考資料

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2) Dynamic Analysis

6.2.2 データストレージの利用

6.2.2.1 SharedPreferences の利用

SharedPreferences.Editor を使って int/boolean/long 値を読み書きする場合、そのデータがオーバーライドされているかどうかをチェックすることはできない。ただし、値を連結する以外の実際の攻撃にはほとんど使用できない。(例えば、制御フローを引き継ぐような追加のエクスプロイトをパックすることはできない) String や StringSet の場合、データがどのように解釈されるかに注意する必要がある。リフレクションベースの永続化を使用しているかについて、Android の「オブジェクトの永続性のテスト」のセクションをチェックして、どのように検証すべきかを確認する。SharedPreferences.Editor を使用して、証明書やキーを保存したり読み取ったりしているかについて、Bouncy Castle で見つかったような脆弱性を考慮し、セキュリティプロバイダのパッチを適用していることを確認する。

どのような場合でも、コンテンツに HMAC をかけることで、追加や変更が加えられていないことを確認することができる。

参考資料

- owasp-mastg Testing Local Storage for Input Validation (MSTG-PLATFORM-2) Using Shared Preferences

ルールブック

- *SharedPreferences* の利用時の入力検証に関する注意事項（必須）

6.2.2.2 SharedPreferences 以外の利用

SharedPreferences.Editor 以外のパブリックストレージメカニズムを使用する場合は、ストレージメカニズムからデータを読み込む瞬間に検証する必要がある。

参考資料

- owasp-mastg Testing Local Storage for Input Validation (MSTG-PLATFORM-2) Using Other Storage Mechanisms

ルールブック

- *SharedPreferences* 以外のパブリックストレージメカニズムを使用する場合の入力検証に関する注意事項（必須）

6.2.3 インジェクション欠陥

Android アプリは、ディープリンク (Intent の一部) を通じて機能を公開することができる。これらのアプリは、以下に対して機能を公開することができる。

- 他のアプリ (ディープリンクや Intent や BroadcastReceiver などの他の IPC メカニズム経由)
- ユーザ (ユーザインターフェース経由)

これらのソースからの入力はどれも信用することができず、検証やサニタイズが必要である。検証は、アプリが期待するデータのみを処理することを保証する。検証を行わない場合、どのような入力もアプリに送ることができ、攻撃者や悪意のあるアプリにアプリの機能を悪用される可能性がある。

アプリの機能が公開されている場合は、ソースコードの以下の部分をチェックする必要がある。

- ディープリンク：テストケース「Testing Deep Links」でも、さらなるテストシナリオを確認する。
- IPC メカニズム (intent 、Binder 、Android 共有メモリ、または BroadcastReceivers) : テストケース「 Test for Sensitive Functionality Exposure Through IPC 」でも、さらなるテストシナリオを確認する。
- ユーザインターフェース : テストケース「 Testing for Overlay Attacks 」を確認する。

脆弱な IPC メカニズムの例を以下に示す。

ContentProviders を使ってデータベース情報にアクセスし、サービスがデータを返すかどうかを調べることができる。データが適切に検証されていない場合、他のアプリが ContentProvider とやりとりしている間に、ContentProvider は SQL インジェクションを受けやすくなるかもしれない。以下の脆弱な ContentProvider の実装を参照する。

```
<provider
    android:name=".OMTG_CODING_003_SQL_Injection_Content_Provider_Implementation"
    android:authorities="sg.vp.owasp_mobile.provider.College">
</provider>
```

上記の AndroidManifest.xml は、エクスポートされるため、他のすべてのアプリが利用できる ContentProvider を定義している。OMTG_CODING_003_SQL_Injection_Content_Provider_Implementation.java クラスのクエ

リ関数を調べる必要がある。

```
@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;
}
```

ユーザが content://sg.vp.owasp_mobile.provider.College/students で STUDENT_ID を提供している間、クエリステートメントは SQL インジェクションを受けやすくなる。もちろん、SQL インジェクションを避けるために準備済みステートメントを使用しなければなりませんが、アプリが期待する入力だけが処理されるように、入力検証も適用されなければならない。

UI から入ってくるデータを処理するアプリの関数は、すべて入力検証を実装する必要がある。

- ユーザインターフェースの入力には、Android Saripaar v2 が使用できる。
- IPC や URL スキームからの入力に対しては、バリデーション関数を作成する。例えば、以下のように、文字列が英数字かどうかを判定する。

```
public boolean isAlphaNumeric(String s){
    String pattern= "^[a-zA-Z0-9]*$";
    return s.matches(pattern);
}
```

検証関数に代わるものとして、例えば整数値のみが期待される場合は Integer.parseInt で型変換を行うことができる。OWASP Input Validation Cheat Sheet に、このトピックに関する詳細情報が含まれている。

参考資料

- owasp-mastg Testing for Injection Flaws (MSTG-PLATFORM-2) Overview

ルールブック

- **ディープリンクを確認し *Web* サイトの正しい関連性を検証する（必須）**
- **セキュリティを考慮し *IPC* メカニズムを使用する（必須）**
- ***UI* から入ってくるデータを処理するアプリの関数は、すべて入力検証を実装する（必須）**

6.2.3.1 動的解析

例えば、ローカルな SQL インジェクションの脆弱性が確認された場合、テスト者は OR 1=1-- のような文字列で入力フィールドを手動でテストする必要がある。

root 化されたデバイスでは、ContentProvider からデータを問い合わせるために、content コマンドを使うことができる。次のコマンドは、上記の脆弱な関数に問い合わせるものである。

```
# content query --uri content://sg.vp.owasp_mobile.provider.College/students
```

SQL インジェクションは、以下のコマンドで悪用される可能性がある。Bob のみのレコードを取得する代わりに、すべてのデータを取得することができる。

```
# content query --uri content://sg.vp.owasp_mobile.provider.College/students --
↪where "name='Bob') OR 1=1--''"
```

参考資料

- owasp-mastg Testing for Injection Flaws (MSTG-PLATFORM-2) Dynamic Analysis

6.2.4 Fragment でのインジェクション

Android SDK は、Preferences activity をユーザに提示する方法を開発者に提供し、開発者がこの抽象クラスを拡張して適応させることを可能にする。なお、API level 29 で非推奨になった。

この抽象クラスは、intent の追加データフィールド、特に PreferenceActivity.EXTRA_SHOW_FRAGMENT(:android:show_fragment) と PreferenceActivity.EXTRA_SHOW_FRAGMENT_ARGUMENTS(:android:show_fragment_arguments) フィールドを解析する。

最初のフィールドには Fragment のクラス名が、2 番目のフィールドには Fragment に渡される input bundle が含まれる。

PreferenceActivity は Reflection を使用して Fragment をロードするため、パッケージまたは Android SDK の内部で任意のクラスがロードされる可能性がある。ロードされたクラスは、この activity をエクスポートするアプリケーションの context で実行される。

この脆弱性を利用すると、攻撃者はターゲットアプリケーションの内部で Fragment を呼び出したり、他のクラスのコンストラクタに存在するコードを実行したりすることができる。Fragment クラスを継承しない intent で渡されたクラスは java.lang.CastException を引き起こしますが、例外が発生する前に空のコンストラクタが実行され、そのクラスのコンストラクタに存在するコードを実行することが可能である。

この脆弱性を防ぐため、Android 4.4 (API level 19) で isValidFragment という新しいメソッドが追加された。これにより、開発者はこのメソッドをオーバーライドし、この context で使用可能な Fragment を定義することができる。

デフォルトの実装では、Android 4.4 (API level 19) より古いバージョンでは true を返し、それ以降のバージョンでは例外を throw する。

参考資料

- owasp-mastg Testing for Injection Flaws (MSTG-PLATFORM-2) Testing for Fragment Injection (MSTG-PLATFORM-2)

6.2.4.1 静的解析

手順 :

- android:targetSdkVersion が 19 未満かどうかを確認する。
- PreferenceActivity クラスを継承するエクスポートされた Activity を検索する。
- メソッド isValidFragment がオーバーライドされているかどうかを判断する。
- 現在、アプリがマニフェストで android:targetSdkVersion を 19 未満の値に設定しており、脆弱なクラスが isValidFragment の実装を含んでいない場合、脆弱性は PreferenceActivity から継承される。
- 修正するには、開発者は android:targetSdkVersion を 19 以上に更新する必要がある。または、android:targetSdkVersion を更新できない場合は、isValidFragment を実装する。

次の例では、このアクティビティを拡張したアクティビティを示す。

```
public class MyPreferences extends PreferenceActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

次の例では、MyPreferenceFragment のみの読み込みを許可する実装で isValidFragment メソッドをオーバーライドしている。

```
@Override
protected boolean isValidFragment(String fragmentName)
{
return "com.fullpackage.MyPreferenceFragment".equals(fragmentName);
}
```

参考資料

- owasp-mastg Testing for Fragment Injection (MSTG-PLATFORM-2) Static Analysis

6.2.4.2 脆弱なアプリと悪用の例

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;
    }
}
```

この脆弱なアクティビティを悪用するには、次のコードを使用してアプリケーションを作成する。

```
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);
```

Vulnerable App と Exploit PoC App はダウンロード可能である。

参考資料

- owasp-mastg Testing for Fragment Injection (MSTG-PLATFORM-2) Example of Vulnerable App and Exploitation

6.2.5 WebView での URL 読み込み

WebView は、Android の組み込みコンポーネントで、アプリケーション内で Web ページを開くことを可能にする。モバイルアプリに関連する脅威に加えて、WebView は一般的な Web の脅威 (例：XSS 、Open Redirect など) にもアプリをさらす可能性がある。

WebView をテストする際に最も重要なことの 1 つは、信頼できるコンテンツのみがロードできることを確認することである。新しく読み込まれるページは、悪意のある可能性があり、WebView のバインディングを悪用しようとしたり、ユーザを騙そうとしたりする可能性がある。ブラウザアプリを開発しているのでなければ、通常は読み込まれるページをアプリのドメインに制限する。そのためには、ユーザが WebView 内に URL を入力したり (Android ではこれがデフォルト) 、信頼されたドメインの外を移動したりする機会さえも与えないようにするのがよい方法である。信頼できるドメイン内を移動する場合でも、ユーザが信頼できないコンテンツへのリンクに遭遇してクリックするリスクはある (例えば、他のユーザがコメントを投稿できるページの場合) 。さらに、開発者によっては、ユーザにとって危険な可能性のあるデフォルトの動作を上書きしてしまうかもしれない。詳しくは、以下の「静的解析」のセクションを確認する。

より安全なウェブブラウジングを提供するために、Android 8.1 (API level 27) では、SafeBrowsing API が導入され、Google が既知の脅威として分類した URL をアプリケーションが検出することができるようになった。

デフォルトでは、WebView はユーザにセキュリティリスクに関する警告を表示し、その URL を読み込むか、ページの読み込みを停止するかを選択できる。SafeBrowsing API を使用すると、脅威をセーフブラウジングに報告するか、既知の脅威に遭遇するたびに安全に戻るなどの特定のアクションを実行することによって、アプリケーションの動作をカスタマイズすることができる。使用例については、Android Developers のドキュメントを確認すること。

SafeBrowsing Network Protocol v4 のクライアントを実装した SafetyNet ライブラリを使用すると、WebViews から独立して SafeBrowsing API を使用できる。SafetyNet では、アプリが読み込む予定のすべての URL を分析できる。セーフブラウジングは URL スキームに依存しないため、異なるスキーム (http 、file など) の URL をチェックし、TYPE_POTENTIALLY_HARMFUL_APPLICATION および TYPE_SOCIAL_ENGINEERING 脅威タイプに対抗することが可能である。

Virus Total は、既知の脅威について URL とローカルファイルを分析するための API を提供する。API リファレンスは Virus Total の開発者ページで入手できる。

既知の脅威をチェックするために URL やファイルを送信する場合、ユーザのプライバシーを侵害する可能性のある機密データが含まれていないこと、またはアプリケーションの機密コンテンツが公開されていないことを確認する。

参考資料

- owasp-mastg Testing for URL Loading in WebViews (MSTG-PLATFORM-2)

ルールブック

- *WebView* は信頼できるコンテンツのみロードできること（必須）

6.2.5.1 静的解析

前述したように、ページナビゲーションの処理については、特にユーザが信頼できる環境からナビゲートできる可能性がある場合、慎重に分析する必要がある。Android のデフォルトかつ最も安全な動作は、ユーザが WebView 内でクリックする可能性のあるリンクはすべてデフォルトの Web ブラウザが開くようにすることである。ただし、このデフォルトのロジックは、ナビゲーション要求をアプリ自体で処理できるようにする WebViewClient を設定することで変更することができる。この場合、次のインターセプトコールバック関数を検索して調べる。

- shouldOverrideUrlLoading は、アプリケーションが疑わしいコンテンツを持つ WebView の読み込みを中止するために true を返すか、WebView が URL を読み込むことを許可するために false を返すかを選択できるようにする。

  考慮すべき点 :

  - このメソッドは POST リクエストでは呼び出されない。
  - このメソッドは XmlHttpRequests 、iFrame 、HTML や <script> タグに含まれる src 属性のために呼び出されない。代わりに、shouldInterceptRequest がこれを処理する必要がある。

- shouldInterceptRequest は、アプリケーションがリソース要求からデータを返すことを可能にする。戻り値が NULL の場合、WebView は通常通りリソースの読み込みを継続する。それ以外の場合は、shouldInterceptRequest メソッドによって返されたデータが使用される。

  考慮すべき点 :

  - このコールバックは、ネットワーク経由でリクエストを送信するスキームだけでなく、さまざまな URL スキーム (http(s): 、data: 、file: など) に対して呼び出される。
  - これは、javascript: または blob: URL、または file:///android_asset/ または file:///android_res/ URL を介してアクセスされるアセットに対しては呼び出されない。リダイレクトの場合、これは最初のリソース URL に対してのみ呼び出され、それ以降のリダイレクト URL は呼び出されない。
  - セーフブラウジングが有効な場合、これらの URL はセーフブラウジングチェックを受けますが、開発者は setSafeBrowsingWhitelist で URL を許可するか、onSafeBrowsingHit コールバックで警告を無視することができる。

このように、WebViewClient を設定した WebView のセキュリティをテストする際に考慮すべき点が多くあるため、WebViewClient のドキュメントをよく読んで理解するようにする。

EnableSafeBrowsing のデフォルト値は true であるが、アプリケーションによっては無効にしている場合がある。セーフブラウジングが有効であることを確認するには、AndroidManifest.xml ファイルを調べ、以下の設定がないことを確認する。

```
<manifest>
    <application>
        <meta-data android:name="android.webkit.WebView.EnableSafeBrowsing"
                   android:value="false" />
        ...
    </application>
</manifest>
```

参考資料

- owasp-mastg Testing for Injection Flaws (MSTG-PLATFORM-2) Static Analysis

ルールブック

- **ユーザが *WebView* 内でクリックする可能性のあるリンクはすべてデフォルトの *Web* ブラウザが開くようにする（推奨）**

6.2.5.2 動的解析

ディープリンクを動的にテストする便利な方法は、Frida または frida-trace を使用し、アプリを使用して WebView 内のリンクをクリックしながら shouldOverrideUrlLoading および shouldInterceptRequest メソッドをフックすることである。また、getHost 、getScheme 、getPath などの関連する Uri メソッドもフックする。これらは通常、リクエストを検査し、既知のパターンや拒否リストに一致させるために使用される。

参考資料

- owasp-mastg Testing for Injection Flaws (MSTG-PLATFORM-2) Dynamic Analysis

6.2.6 ルールブック

1. ***WebView* を確認して、アプリからレンダリングされた信頼できない入力がないか調査する（必須）**
2. ***SharedPreferences* の利用時の入力検証に関する注意事項（必須）**
3. ***SharedPreferences* 以外のパブリックストレージメカニズムを使用する場合の入力検証に関する注意事項（必須）**
4. ***UI* から入ってくるデータを処理するアプリの関数は、すべて入力検証を実装する（必須）**
5. ***WebView* は信頼できるコンテンツのみロードできること（必須）**
6. **ユーザが *WebView* 内でクリックする可能性のあるリンクはすべてデフォルトの *Web* ブラウザが開くようにする（推奨）**

6.2.6.1 WebView を確認して、アプリからレンダリングされた信頼できない入力がないか調査する（必須）

WebView を確認して、アプリからレンダリングされた信頼できない入力がないか調査する必要がある。

WebView によって開かれた URL の一部がユーザの入力によって決定される場合、XSS 問題が存在する可能性がある。

以下サンプルコードは、Linus Särud 氏によって報告された Zoho Web Service における XSS 問題である。

Java での例：

```
webView.loadUrl("javascript:initialize(" + myNumber + ");");
```

Kotlin での例：

```
webView.loadUrl("javascript:initialize($myNumber);")
```

XSS 問題を回避するためには、webView.loadUrl で "myNumber" を使用する前に、"myNumber" の入力チェックを行う。

また、以下サンプルコードは、ユーザ入力によって決まる XSS 問題のもう一つの例のパブリックオーバーライドされたメソッドである。

Java での例：

```
@Override
public boolean shouldOverrideUrlLoading(WebView view, String url) {
  if (url.substring(0,6).equalsIgnoreCase("yourscheme:")) {
    // parse the URL object and execute functions
  }
}
```

Kotlin での例：

```
    fun shouldOverrideUrlLoading(view: WebView, url: String): Boolean {
        if (url.substring(0, 6).equals("yourscheme:", ignoreCase = true)) {
            // parse the URL object and execute functions
        }
    }
```

XSS 問題を回避するためには、"url" を使用する前に "url" の入力チェックを行う。

以下のベストプラクティスに従っていることを確認する。

- HTML 、JavaScript 、その他の解釈されるコンテキストでは、絶対に必要な場合を除き、信頼できないデータはレンダリングされない。
- エスケープ文字には、HTML のエンティティエンコーディングなど、適切なエンコーディングが適用される。注：HTML が他のコードにネストされている場合、エスケープのルールは複雑になる。例えば、JavaScript ブロックの中にある URL を表示させる場合などである。

レスポンスでデータがどのようにレンダリングされるかを検討する。例えば、データが HTML コンテキストでレンダリングされる場合、エスケープする必要がある 6 つの制御文字が存在する。

表 6.2.6.1.1　エスケープする必要がある制御文字一覧

Character	Escaped
&	&amp;
<	&lt;
>	&gt;
"	&quot;
'	&#x27;
/	&#x2F;

エスケープルールやその他の防止策の包括的なリストについては、OWASP XSS Prevention Cheat Sheet を参照する。

これに違反する場合、以下の可能性がある。

- XSS 問題が存在する可能性がある。

6.2.6.2 SharedPreferences の利用時の入力検証に関する注意事項（必須）

SharedPreferences をデータストレージとして利用する場合、以下に注意する必要がある。

- String や StringSet の場合、データがどのように解釈されるかに注意する。SharedPreferences で String 使用時のサンプルコード：

```
// 設定値 String を保存
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();
}

// 設定値 String を取得
public static String loadString(Context ctx, String key) {
    SharedPreferences prefs = ctx.getSharedPreferences(APP_NAME, Context.MODE_
↪PRIVATE);
    return prefs.getString(key, "");
}
```

SharedPreferences で StringSet 使用時のサンプルコード：

```
// 設定値 Set<String> を保存
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();
}

// 設定値 Set<String> を取得
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>());
}
```

- リフレクションベースの永続化を使用しているかについて、Android の「オブジェクトの永続性のテスト」のセクションをチェックして、どのように検証すべきかを確認する。
- SharedPreferences.Editor を使用して、証明書やキーを保存したり読み取ったりしているかについて、

Bouncy Castle で見つかったような脆弱性を考慮し、セキュリティプロバイダのパッチを適用していることを確認する。

これに違反する場合、以下の可能性がある。

- 文字列が誤ったデータ型で解釈される。

6.2.6.3 SharedPreferences 以外のパブリックストレージメカニズムを使用する場合の入力検証に関する注意事項（必須）

SharedPreferences 以外のパブリックストレージメカニズムを使用する場合は、データがストレージメカニズムから読み取られた時点で入力を検証する必要がある。

以下サンプルコードは、データストレージからファイルを読み取る処理の一例。

```
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;
}
```

上記サンプルコードのように安全にデータを読み取る場合は、"text" を返却する前に "text" を検証する。

これに違反する場合、以下の可能性がある。

- 意図しない入力として読み取られる可能性がある。

6.2.6.4 UI から入ってくるデータを処理するアプリの関数は、すべて入力検証を実装する（必須）

UI から入ってくるデータを処理するアプリの関数は、すべて入力検証を実装する必要がある。

- ユーザインターフェースの入力には、Android Saripaar v2 が使用できる。

  ※最終リリースが 2015/9/18 のためサンプルコード省略

- IPC や URL スキームからの入力に対しては、バリデーション関数を作成する。

  以下サンプルコードは、文字列が英数字かどうかを判定処理の一例。

```
public boolean isAlphaNumeric(String s){
    String pattern= "^[a-zA-Z0-9]*$";
    return s.matches(pattern);
}
```

これに違反する場合、以下の可能性がある。

- 意図しない入力として読み取られる可能性がある。

6.2.6.5 WebView は信頼できるコンテンツのみロードできること（必須）

WebView は、Android の組み込みコンポーネントで、アプリケーション内で Web ページを開くことを可能にする。モバイルアプリに関連する脅威に加えて、WebView は一般的な Web の脅威 (例：XSS 、Open Redirect など) にもアプリをさらす可能性がある。そのため、信頼できるコンテンツのみがロードできるようにする必要がある。

安全なウェブブラウジング

より安全なウェブブラウジングを行うため、Android 8.1 (API level 27) で導入された SafeBrowsing API を使用する。これを使用することで、Google が既知の脅威として分類した URL をアプリケーションが検出することができる。デフォルトでは、既知の脅威をユーザに警告するインタースティシャルが WebView によって表示される。

以下のサンプルコードは、既知の脅威を検出したときに、アプリの WebView インスタンスに対して安全なページに必ず戻るよう指示する処理の一例。

以下は AndroidManifest での宣言。

```
<manifest>
    <application>
        ...
        <meta-data android:name="android.webkit.WebView.EnableSafeBrowsing"
                   android:value="true" />
    </application>
</manifest>
```

以下は WebView 呼び出しクラスのコールバック処理。

```
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!")
            }
        })
    }
}
```

以下はアクセスした URL が安全でないと判断された際に呼び出される コールバック onSafeBrowsingHit での処理。

```
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()
    }
}
```

また、SafeBrowsing Network Protocol v4 のクライアントを実装した SafetyNet ライブラリを使用すると、WebViews から独立して SafeBrowsing API を使用できる。SafetyNet では、アプリが読み込む予定のすべての URL を分析できる。セーフブラウジングは URL スキームに不可知論的であるため、異なるスキーム (http 、file など) の URL をチェックし、TYPE_POTENTIALLY_HARMFUL_APPLICATION および TYPE_SOCIAL_ENGINEERING 脅威タイプに対抗することが可能である。

以下のサンプルコードは SafetyNet ライブラリによる URL チェックをリクエストする処理の一例。

```
SafetyNet.getClient(this).lookupUri(
        url,
        SAFE_BROWSING_API_KEY,
        SafeBrowsingThreat.TYPE_POTENTIALLY_HARMFUL_APPLICATION,
        SafeBrowsingThreat.TYPE_SOCIAL_ENGINEERING
)
        .addOnSuccessListener(this) { sbResponse ->
            // サービスとの通信成功
            if (sbResponse.detectedThreats.isEmpty()) {
                // 脅威が見つからなかった場合
            } else {
                // 威が見つかった場合
            }
        }
        .addOnFailureListener(this) { e: Exception ->
            // サービスとの通信失敗
            if (e is ApiException) {
                // Google Play Services API でのエラーが発生した場合
            } else {
                // 別のエラーが発生した場合
            }
        }
```

その他に、Virus Total を利用することで、既知の脅威について URL とローカルファイルを分析することができる。

また、ブラウザアプリを開発しているのでなければ、通常は読み込まれるページをアプリのドメインに制限する。そのためには、ユーザが WebView 内に URL を入力したり (Android ではこれがデフォルト) 、信頼されたドメインの外を移動したりする機会さえも与えないようにするのがよい方法である。信頼できるドメイン内を移動する場合でも、ユーザが信頼できないコンテンツへのリンクに遭遇してクリックするリスクはある (例えば、他のユーザがコメントを投稿できるページの場合) 。

これに違反する場合、以下の可能性がある。

- Web の脅威 (例：XSS 、Open Redirect など) に脆弱になる。

6.2.6.6 ユーザが WebView 内でクリックする可能性のあるリンクはすべてデフォルトの Web ブラウザが開くようにする（推奨）

Android のデフォルトかつ最も安全に動作させるために、ユーザが WebView 内でクリックする可能性のあるリンクはすべてデフォルトの Web ブラウザが開くようにする。

以下に WebView 内でクリックされたリンクをデフォルトの Web ブラウザで開くサンプルコードを示す。

```
  @Override
  public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest␣
↪request) {
    Uri uri = request.getUri();
    String uriPath = uri.getPath();
```

(次のページに続く)

(前のページからの続き)

```
    if(uriPathCheck(uriPath)){
      Intent intent = new Intent(Intent.ACTION_VIEW, uri)
      view.getContext().startActivity(intent);
      return true;
    }

    return false;
  }
```

ただし、このデフォルトのロジックは、ナビゲーション要求をアプリ自体で処理できるようにする WebViewClient を設定することで変更することができる。

そのため、以下インターセプトコールバック関数について考慮すべき点を考慮すること。

- shouldOverrideUrlLoading は、アプリケーションが疑わしいコンテンツを持つ WebView の読み込みを中止するために true を返すか、WebView が URL を読み込むことを許可するために false を返すかを選択できるようにする。

  考慮すべき点 :

  - このメソッドは POST リクエストでは呼び出されない。
  - このメソッドは XmlHttpRequests 、iFrame 、HTML や <script> タグに含まれる src 属性のために呼び出されない。代わりに、shouldInterceptRequest がこれを処理する必要がある。

- shouldInterceptRequest は、アプリケーションがリソース要求からデータを返すことを可能にする。戻り値が NULL の場合、WebView は通常通りリソースの読み込みを継続する。それ以外の場合は、shouldInterceptRequest メソッドによって返されたデータが使用される。

  考慮すべき点 :

  - このコールバックは、ネットワーク経由でリクエストを送信するスキームだけでなく、さまざまな URL スキーム (http(s)：、data：、file：など) に対して呼び出される。
  - これは、javascript: または blob: URL、または file:///android_asset/ または file:///android_res/ URL を介してアクセスされるアセットに対しては呼び出されない。リダイレクトの場合、これは最初のリソース URL に対してのみ呼び出され、それ以降のリダイレクト URL は呼び出されない。
  - セーフブラウジングが有効な場合、これらの URL はセーフブラウジングチェックを受けますが、開発者は setSafeBrowsingWhitelist で URL を許可するか、onSafeBrowsingHit コールバックで警告を無視することができる。

また、EnableSafeBrowsing を true（デフォルト値）に設定することで、セーフブラウジングを有効に設定できる。設定値を false に指定することで、セーフブラウジングが無効となることに注意する。

以下は AndroidManifest.xml での EnableSafeBrowsing の設定例。

```
<manifest>
    <application>
        <meta-data android:name="android.webkit.WebView.EnableSafeBrowsing"
```

(次のページに続く)

(前のページからの続き)

```
                android:value="true" />
        ...
    </application>
</manifest>
```

これに違反する場合、以下の可能性がある。

- 意図しないコンテンツをアプリケーションに読み込まれる可能性がある。
- 悪意のある JavaScript をアプリケーション上で実行する可能性がある。

6.3 MSTG-PLATFORM-3

アプリはメカニズムが適切に保護されていない限り、カスタム URL スキームを介して機密な機能をエクスポートしていない。

6.3.1 ディープリンクの種類

ディープリンクは、ユーザをアプリ内の特定のコンテンツに直接移動させる任意のスキームの URI である。アプリは、Android Manifest に intent filters を追加し、受信した intent からデータを抽出して、ユーザを正しい activity にナビゲートすることで、ディープリンクを設定することができる。

Android は、次の 2 種類のディープリンクをサポートしている。

- カスタム URL スキーム：任意のカスタム URL スキーム、たとえば myapp:// (OS によって検証されない) を使用するディープリンクである。
- Android アプリリンク (Android 6.0 (API level 23) 以上) は、http:// および https:// スキームを使用し、autoVerify 属性 (OS による検証を開始する) を含むディープリンクである。

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Overview

6.3.2 ディープリンクの未検証による競合

検証されていないディープリンクを使用すると、重大な問題が発生する。ユーザのデバイスにインストールされている他のアプリが、同じ intent を宣言して処理しようとする可能性があり、これはディープリンクの衝突として知られている。任意のアプリケーションは、他のアプリケーションに属する全く同じディープリンクの制御を宣言することができる。

最近の Android では、ディープリンクを処理するアプリケーションを選択するよう求める、いわゆる曖昧さ回避ダイアログがユーザに表示されるようになった。ユーザは、正規のアプリケーションではなく、悪意のあるアプリケーションを選択してしまう可能性がある。

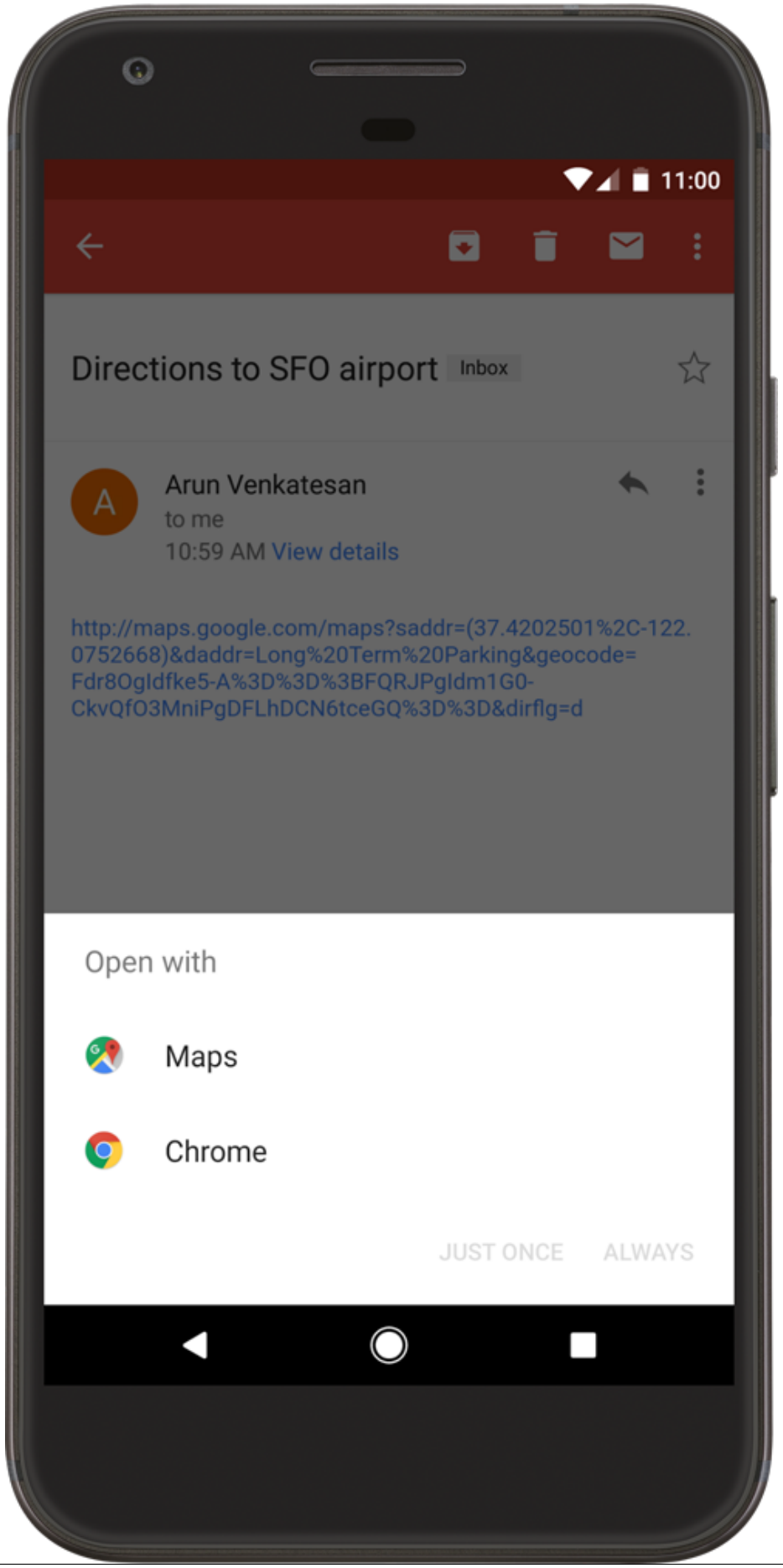


図 6.3.2.1 曖昧さ回避ダイアログ

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Deep Link Collision

6.3.3 Android アプリのリンク

ディープリンクの衝突問題を解決するために、Android 6.0 (API Level 23) では、開発者が明示的に登録した Web サイトの URL を元に検証されたディープリンクである Android アプリリンク が導入された。App Link をクリックすると、アプリがインストールされている場合は、すぐにそのアプリが開く。

未検証のディープリンクとの主な違いは以下の通りである。

- App Links は、http:// と https:// スキームのみを使用し、その他のカスタム URL スキームは許可されない。
- App Links は、Digital Asset Links file を HTTPS 経由で提供するために、ライブドメインが必要である。
- App Links は、ユーザが開いたときに曖昧さ回避のダイアログが表示されないため、ディープリンクの衝突は発生しない。

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Android App Links

6.3.4 ディープリンクのテスト

既存のディープリンク (アプリリンクを含む) は、アプリの攻撃対象領域を拡大する可能性がある。これには、リンクのハイジャック、機密機能の漏洩など、多くのリスクが含まれる。また、アプリが動作する Android のバージョンもリスクに影響する。

- Android 12 (API level 31) 以前は、アプリに検証不可能なリンクがある場合、システムがそのアプリのすべての Android アプリリンクを検証しない原因となることがあった。
- Android 12 (API level 31) 以降では、アプリは攻撃面が減少するというメリットがある。一般的な Web intent は、ターゲット アプリがその Web intent に含まれる特定のドメインに対して承認されていない限り、ユーザの既定のブラウザアプリに解決される。

すべてのディープリンクを列挙し、Web サイトの正しい関連性を検証する必要がある。特に入力データは信頼できないと判断されるため、常に検証する必要がある。

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Testing Deep Links

ルールブック

- **ディープリンクを確認し *Web* サイトの正しい関連性を検証する（必須）**

6.3.5 静的解析

6.3.5.1 ディープリンクの列挙

Android Manifest の検査 :

apktool を使用してアプリをデコードし、Android Manifest ファイルを検査して <intent-filter> 要素を探せば、ディープリンク (カスタム URL スキームあり/なし) が定義されているかどうかを簡単に判断することができる。

- カスタム URL スキーム : 次の例では、myapp:// というカスタム URL スキームでディープリンクを指定している。

```
<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>
```

- ディープリンク : 次の例では、http:// と https:// の両方のスキームと、それを有効にするホストとパス (この場合、完全な URL は https://www.myapp.com/my/app/path) を使用して、ディープリンクを指定している。

```
<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>
```

- アプリリンク : <intent-filter> に android:autoVerify="true" というフラグが含まれている場合、Android システムは、アプリリンクを検証するためにデジタルアセットリンクファイルにアクセスしようと宣言された android:host にアクセスするようになる。ディープリンクは、検証が成功した場合にのみ、アプリリンクとみなされる。

```
<intent-filter android:autoVerify="true">
```

ディープリンクを記載する場合、同じ <intent-filter> 内の <data> 要素は、その組み合わせ属性のすべてのバリエーションを考慮し、実際にはマージされることに注意する。

```
<intent-filter>
  ...
  <data android:scheme="https" android:host="www.example.com" />
  <data android:scheme="app" android:host="open.my.app" />
</intent-filter>
```

https://www.example.com と app://open.my.app のみに対応しているように思われるかもしれないが実際には以下をサポートしている。

- https://www.example.com
- app://open.my.app
- app://www.example.com
- https://open.my.app

Dumpsys の使用 : adb を使用して以下のコマンドを実行すると、すべてのスキームが表示される。

```
adb shell dumpsys package com.example.package
```

Android「 App Link Verification 」テスターの使用 : Android の「 App Link Verification 」テスタースクリプトを使用して、すべてのディープリンク (list-all) またはアプリリンク (list-applinks) のみをリストアップする。

```
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
```

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Enumerate Deep Links

ルールブック

- **ディープリンクを確認し *Web* サイトの正しい関連性を検証する（必須）**

6.3.5.2 Web サイトへの正常な関連付けの検証

ディープリンクに android:autoVerify="true" 属性が含まれている場合でも、アプリリンクと見なされるには実際に検証する必要がある。完全な検証を妨げる可能性のある設定ミスがないかをテストする必要がある。

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Check for Correct Website Association

ルールブック

- **ディープリンクを確認し *Web* サイトの正しい関連性を検証する（必須）**

自動検証

Android の「 App Link Verification 」テスタースクリプトを使用して、すべてのアプリリンクの検証状況を取得する (verify-applinks) 。例はこちらを確認する。

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Static Analysis Automatic Verification

Android 12 (API level 31) 以上でのみ使用可能

アプリが Android 12 (API level 31) を対象としているかどうかに関係なく、adb を使用して検証ロジックをテストすることができる。この機能により、次のことが可能になる。

- 検証プロセスを手動で呼び出す。
- 対象アプリの Android アプリリンクの状態をデバイス上でリセットする。
- ドメイン認証プロセスを起動する。

また、検証結果を確認することもできる。以下のような例がある。

```
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
```

adb shell dumpsys package com.example.package を実行しても同じ情報が得られる（Android 12 (API level 31) 以降のみ）。

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Automatic Verification

手動検証

このセクションでは、検証プロセスが失敗した、または実際にトリガーされなかった理由について、いくつかの理由を詳しく説明する。詳細については、Android Developers Documentation およびホワイトペーパー「Measuring the Insecurity of Mobile Deep Links of Android」を参照する。

デジタルアセットリンクファイルの確認 :

- 欠落しているデジタルアセットリンクファイルを確認する。
  - ドメインの /.well-known/ パスで検索する。例 : https://www.example.com/.well-known/assetlinks.json
  - または、https://digitalassetlinks.googleapis.com/v1/statements:list?source.web.site=www.example.com を試す。
- HTTP 経由で提供される有効なデジタルアセットリンクファイルを確認する。
- HTTPS 経由で提供される無効なデジタルアセットリンクファイルを確認する。
  - ファイルに不正な JSON が含まれている。

- ファイルにターゲットアプリのパッケージが含まれていない。

アプリのセキュリティを強化するため、サーバが http://example.com から https://example.com や example.com から www.example.com などのリダイレクトを設定している場合、システムはアプリの Android アプリリンクを一切検証しない。

intent filter が、異なるサブドメインを持つ複数のホストをリストアップする場合、各ドメインに有効なデジタルアセットリンクファイルが存在する必要がある。例えば、次の intent filter は、www.example.com と mobile.example.com を受け入れ可能な intent URL ホストとして含める。

```
<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>
```

ディープリンクを正しく登録するためには、有効なデジタルアセットリンクファイルが https://www.example.com/.well-known/assetlinks.json と https://mobile.example.com/.well-known/assetlinks.json の両方で公開されている必要がある。

ホスト名にワイルドカード (*.example.com など) が含まれている場合、ルートホスト名 https://example.com/.well-known/assetlinks.json 、有効なデジタルアセットリンクファイルを見つけることができる。

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Manual Verification

6.3.5.3 Handler Method の検証

ディープリンクが正しく検証されたとしても、Handler Method のロジックは慎重に分析する必要がある。特に、(ユーザや他のアプリによって外部から制御される) データの送信にディープリンクが使用されている場合は注意が必要である。

まず、ターゲットの <intent-filter> を定義する Android Manifest の <activity> 要素から Activity 名を取得し、getIntent と getData の使用方法を検索する。これらのメソッドを見つけるこの一般的なアプローチは、リバースエンジニアリングを行う際にほとんどのアプリケーションで使用でき、アプリケーションがディープリンクをどのように使用し、外部から提供された入力データをどのように処理するか、および何らかの不正行為の対象となり得るかを理解しようとする場合に重要である。なお、getIntent は現在非推奨のため、parseUri の使用が推奨される。

次の例は、jadx で逆コンパイルされた典型的な Kotlin アプリのスニペットである。静的解析から、

com.mstg.deeplinkdemo.WebViewActivity の一部として deeplinkdemo://load.html/ をサポートしていることが分かる。

```
// 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);
            ...
```

deeplink_url 文字列変数をたどるだけで、wv.loadUrl 呼び出しの結果を確認できる。これは、攻撃者が WebView に読み込まれる URL を完全に制御できることを意味する (上記ように JavaScript が有効になっている) 。

同じ WebView が、攻撃者が制御するパラメータをレンダリングしている可能性もある。その場合、以下のディープリンクのペイロードは、WebView のコンテキスト内で Reflected Cross-Site Scripting (XSS) を引き起こす可能性がある。

```
deeplinkdemo://load.html?attacker_controlled=<svg onload=alert(1)>
```

しかし、それ以外にも多くの可能性がある。以下のセクションで、期待されること、さまざまなシナリオをテストする方法について、必ず確認する。

- 「クロスサイトスクリプティングの不具合 (MSTG-PLATFORM-2) 」
- 「インジェクションの不具合 (MSTG-ARCH-2、MSTG-PLATFORM-2) 」
- 「オブジェクトの永続性のテスト (MSTG-PLATFORM-8) 」
- 「 WebView における URL ロードのテスト (MSTG-PLATFORM-2) 」
- 「 WebView における JavaScript 実行のテスト (MSTG-PLATFORM-5) 」
- 「 WebView プロトコルハンドラのテスト (MSTG-PLATFORM-6) 」

さらに、公的な報告書を検索して読むことを推奨する (検索語 : "deep link*"|"deeplink*"site:https://hackerone.com/reports/) 。例は以下である。

- 「 HackerOne#1372667 」ディープリンクからベアラートークンを盗むことが可能
- 「 HackerOne#401793 」安全でないディープリンクで機密情報漏洩につながる

- 「HackerOne#583987」Android アプリのディープリンクからフォローアクションで CSRF を引き起こす
- 「HackerOne#637194」Android アプリで生体認証セキュリティ機能のバイパスが可能
- 「HackerOne#341908」ダイレクトメッセージのディープリンクを利用した XSS

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Check the Handler Method

ルールブック

- **ディープリンクが正しく検証されたとしても *Handler Method* のロジックを慎重に分析する（必須）**

6.3.6 動的解析

ここでは、静的解析から得られたディープリンクのリストを使用して、各 Handler Method と処理されたデータ (存在する場合) を繰り返し、決定する。最初に Frida フックを起動し、それからディープリンクの呼び出しを開始する。

次の例では、deeplinkdemo://load.html というディープリンクを受け入れるターゲットアプリを想定している。しかし、対応する Handler Method も、それが受け入れる可能性のあるパラメータもまだ分かっていない。

「ステップ 1」Frida Hooking: Frida CodeShare にあるスクリプト「Android Deep Link Observer」を使用すると、Intent.getData の呼び出しをトリガーとして呼び出されるすべてのディープリンクを監視できる。また、このスクリプトをベースにして、ユースケースに応じて独自の変更を加えることもできる。今回は、Intent.getData を呼び出すメソッドに興味があるので、スタックトレースをスクリプトに含めた。

「ステップ 2」Invoking Deep Links: これで、adb と Activity Manager (am) を使ってディープリンクを呼び出すことができ、Android デバイス内に intents を送信することができる。例は以下のとおりである。

```
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
```

これにより、http/https スキーマを使用している場合、またはインストールされている他のアプリが同じカスタム URL スキーマをサポートしている場合に、曖昧さ回避ダイアログが表示される可能性がある。パッケージ名を含めることで、明示的な intent にすることができる。

この起動により、以下のようなログが記録される。

```
[*] 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)
```

このケースでは、任意のパラメータ (?message=ok) と fragment(#part1) を含むディープリンクを作成している。それらが使用されているかどうかはまだ不明である。上記の情報から、アプリをリバースエンジニアリングするために今すぐ使える有用な情報が明らかになった。考慮すべき点については、「ハンドラメソッドのチェック」のセクションを参照する。

- ファイル：WebViewActivity.kt
- クラス : com.mstg.deeplinkdemo.WebViewActivity
- メソッド : onCreate

場合によっては、ターゲットアプリとやり取りすることがわかっている他のアプリケーションを利用することもできる。アプリをリバースエンジニアリングする (例えば、すべての文字列を抽出し、ターゲットのディープリンク (前のケースでは deeplinkdemo:///load.html) を含むものをフィルタする) 、あるいは、前に述べたようにアプリをフックしながらそれらをトリガーとして使用することができる。

参考資料

- owasp-mastg Testing Deep Links (MSTG-PLATFORM-3) Dynamic Analysis

6.3.7 ルールブック

1. ディープリンクを確認し *Web* サイトの正しい関連性を検証する（必須）
2. ディープリンクが正しく検証されたとしても *Handler Method* のロジックを慎重に分析する（必須）

6.3.7.1 ディープリンクを確認し Web サイトの正しい関連性を検証する（必須）

既存のディープリンク (アプリリンクを含む) は、アプリの攻撃対象領域を拡大する可能性があり、これにはリンクのハイジャック、機密機能の漏洩など、多くのリスクが含まれる。

上記を防ぐために、すべてのディープリンクを列挙し、Web サイトの正しい関連性を検証する必要がある。特に入力データは信頼できないと判断されるため、常に検証する必要がある。

ディープリンクの列挙

AndroidManifest ファイルを検査して 要素から、ディープリンク (カスタム URL スキームあり/なし) が定義されているかどうかを判断する。

以下はディープリンクの定義例。

- カスタム URL スキーム：次の例では、myapp:// というカスタム URL スキームでディープリンクを指定している。

```
<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>
```

- ディープリンク：次の例では、http:// と https:// の両方のスキームと、それを有効にするホストとパス (この場合、完全な URL は https://www.myapp.com/my/app/path) を使用して、ディープリンクを指定している。

```
<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>
```

- アプリリンク：<intent-filter> に android:autoVerify="true" というフラグが含まれている場合、Android システムは、アプリリンクを検証するためにデジタルアセットリンクファイルにアクセスしようと宣言された android:host にアクセスするようになる。ディープリンクは、検証が成功した場合にのみ、アプリリンクとみなされる。

```
<intent-filter android:autoVerify="true">
```

ディープリンクを記載する場合、同じ <intent-filter> 内の <data> 要素は、その組み合わせ属性のすべてのバリエーションを考慮し、実際にはマージされることに注意する。

```
<intent-filter>
  ...
  <data android:scheme="https" android:host="www.example.com" />
  <data android:scheme="app" android:host="open.my.app" />
</intent-filter>
```

上記の定義では、実際には以下をサポートしている。

- https://www.example.com
- app://open.my.app
- app://www.example.com
- https://open.my.app

また、ディープリンクに android:autoVerify="true" 属性が含まれている場合でも、アプリリンクと見なされるには実際に検証する必要がある。完全な検証を妨げる可能性のある設定ミスがないかをテストする必要がある。

これに違反する場合、以下の可能性がある。

- リンクのハイジャックや機密機能の漏洩の可能性がある。

6.3.7.2 ディープリンクが正しく検証されたとしても Handler Method のロジックを慎重に分析する（必須）

ディープリンクが正しく検証されたとしても、Handler Method のロジックは慎重に分析する必要がある。特に、(ユーザや他のアプリによって外部から制御される) データの送信にディープリンクが使用されている場合は注意が必要である。

以下のサンプルコードでは、deeplink_url 文字列変数をたどるだけで wv.loadUrl 呼び出しの結果を確認することができるため注意する。

```
// 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) {
```

(次のページに続く)

(前のページからの続き)

```
            WebView wv = (WebView) findViewById;
            wv.getSettings().setJavaScriptEnabled(true);
            if (deeplink_url != null) {
                wv.loadUrl(deeplink_url);
            ...
```

同じ WebView が、攻撃者が制御するパラメータをレンダリングしている可能性もある。その場合、以下のディープリンクのペイロードは、WebView のコンテキスト内で Reflected Cross-Site Scripting (XSS) を引き起こす可能性がある。

```
deeplinkdemo://load.html?attacker_controlled=<svg onload=alert(1)>
```

しかし、それ以外にも多くの可能性がある。以下のセクションで、期待されること、さまざまなシナリオをテストする方法について、必ず確認する。

- 「クロスサイトスクリプティングの不具合 (MSTG-PLATFORM-2) 」
- 「インジェクションの不具合 (MSTG-ARCH-2、MSTG-PLATFORM-2) 」
- 「オブジェクトの永続性のテスト (MSTG-PLATFORM-8) 」
- 「 WebView における URL ロードのテスト (MSTG-PLATFORM-2) 」
- 「 WebView における JavaScript 実行のテスト (MSTG-PLATFORM-5) 」
- 「 WebView プロトコルハンドラのテスト (MSTG-PLATFORM-6) 」

さらに、公的な報告書を検索して読むことを推奨する (検索語 : "deep link*"|"deeplink*"site:https://hackerone.com/reports/) 。例は以下である。

- 「 HackerOne#1372667 」ディープリンクからベアラートークンを盗むことが可能
- 「 HackerOne#401793 」安全でないディープリンクで機密情報漏洩につながる
- 「 HackerOne#583987 」Android アプリのディープリンクからフォローアクションで CSRF を引き起こす
- 「 HackerOne#637194 」Android アプリで生体認証セキュリティ機能のバイパスが可能
- 「 HackerOne#341908 」ダイレクトメッセージのディープリンクを利用した XSS

これに違反する場合、以下の可能性がある。

- リンク先情報が読み取られる可能性がある。

6.4 MSTG-PLATFORM-4

アプリはメカニズムが適切に保護されていない限り、IPC 機構を通じて機密な機能をエクスポートしていない。

6.4.1 IPC による機密機能の公開

モバイルアプリケーションの実装では、開発者は IPC のための伝統的な技術 (共有ファイルやネットワークソケットの使用など) を適用することができる。モバイルアプリケーションプラットフォームが提供する IPC システム機能は、従来の技術よりもはるかに成熟しているため、これを使用する必要がある。セキュリティを考慮せずに IPC メカニズムを使用すると、アプリケーションから機密データが漏洩したり、公開されたりする可能性がある。

以下は、機密データを公開する可能性がある Android IPC メカニズムのリストである。

- Binders
- Services
- Bound Services
- AIDL
- Intents
- Content Providers

参考資料

- owasp-mastgTesting for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Overview

ルールブック

- **セキュリティを考慮し *IPC* メカニズムを使用する（必須）**

6.4.1.1 Activity の公開

AndroidManifest の調査 Sieve アプリでは、<activity > で識別される 3 つのエクスポートされた activity が見つかる。

```
<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" />
```

(次のページに続く)

(前のページからの続き)

```
<activity android:clearTaskOnLaunch="true" android:excludeFromRecents="true"␣
↪android:exported="true" android:finishOnTaskLaunch="true" android:label="@string/
↪title_activity_pwlist" android:name=".PWList" />
```

ソースコードの検査 PWList.java Activity を調べてみると、すべてのキーのリストアップ、追加、削除などのオプションが提供されていることがわかる。これを直接呼び出すと、LoginActivity をバイパスすることができる。これについては、以下の動的解析に記載されている。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Activities

ルールブック

- セキュリティを考慮し *IPC* メカニズムを使用する（必須）

6.4.1.2 Service の公開

AndroidManifest の調査 Sieve アプリでは、<service> で識別される 2 つのエクスポートされた service が見つかる。

```
<service android:exported="true" android:name=".AuthService" android:process=
↪":remote" />
<service android:exported="true" android:name=".CryptoService" android:process=
↪":remote" />
```

ソースコードの検査 android.app.Service というクラスのソースコードを確認する。ターゲットアプリをリバースさせることで、AuthService という service が、ターゲットアプリのパスワード変更と PIN プロテクトの機能を提供していることがわかる。

```
  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.
```

(次のページに続く)

(前のページからの続き)

```
↪mwr.example.sieve.PIN"))) {
                            returnVal = 0;
                        } else {
                            returnVal = 1;
                        }
                    } else {
                        sendUnrecognisedMessage();
                        return;
                    }
            }
    }
```

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Services

ルールブック

- セキュリティを考慮し *IPC* メカニズムを使用する（必須）

6.4.1.3 BroadcastReceiver の公開

AndroidManifest の調査「 Android Insecure Bank 」アプリでは、<receiver> で識別される broadcast receiver がマニフェストに含まれている。

```
<receiver android:exported="true" android:name="com.android.insecurebankv2.
↪MyBroadCastReceiver">
    <intent-filter>
        <action android:name="theBroadcast" />
    </intent-filter>
</receiver>
```

ソースコードの検査 sendBroadcast 、sendOrderedBroadcast 、sendStickyBroadcast のような文字列をソースコードから検索する。アプリケーションが機密データを送信していないことを確認する。

Intent がブロードキャストされ、アプリケーション内でのみ受信される場合、LocalBroadcastManager を使用して、他のアプリがブロードキャストメッセージを受信するのを防ぐことができる。これにより、機密情報が漏洩するリスクが軽減される。

レシーバーの目的をさらに理解するには、静的解析をより深く行い、クラス android.content.BroadcastReceiver と、動的に receiver を作成するために使用する Context.registerReceiver メソッドの使用状況を検索する必要がある。

ターゲットアプリケーションのソースコードの以下の抜粋は、BroadcastReceiver が、ユーザの復号されたパスワードを含む SMS メッセージの送信をトリガーしていることを示している。

```
public class MyBroadCastReceiver extends BroadcastReceiver {
  String usernameBase64ByteString;
  public static final String MYPREFS = "mySharedPreferences";
```

(次のページに続く)

(前のページからの続き)

```
  @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();
                System.out.println("For the changepassword - phonenumber:
↪"+textPhoneno+" password is: "+textMessage);
smsManager.sendTextMessage(textPhoneno, null, textMessage, null, null);
           }
      }
  }
}
```

BroadcastReceiver は android:permission 属性を使用する必要がある。そうしないと、他のアプリケーションがそれらを呼び出すことができる。Context.sendBroadcast (intent, receiverPermission); を使用して、受信者がブロードキャストを読むために持っていなければならないパーミッションを指定することができる。また、この Intent が解決するコンポーネントを制限する、明示的なアプリケーションパッケージ名を設定することもできる。デフォルト値 (null) のままにしておくと、すべてのアプリケーションのすべてのコンポーネントが考慮される。NULL 以外の場合、Intent は指定されたアプリケーションパッケージのコンポーネントにのみマッチする。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Broadcast Receivers

ルールブック

- セキュリティを考慮し *IPC* メカニズムを使用する（必須）
- *Broadcast* の受信側と送信側の両方で権限を必要とする場合があることに注意する（必須）

6.4.2 静的解析

ソースコードに含まれるすべての activities 、services 、および content providers を宣言する必要がある AndroidManifest.xml を調べることから始める (そうしないと、システムはそれらを認識せず、実行されない) 。Broadcast receivers は、マニフェストで宣言するか、動的に作成することができます。次のような要素を識別することが望まれます。

- <intent-filter>
- <service>
- <provider>
- <receiver>

「エクスポートされた」activity 、service 、または content は、他のアプリからアクセスすることができる。エクスポートされたコンポーネントを指定するには、2 つの一般的な方法があります。明らかなのは、export タグを true に設定することである (android:exported="true") 。2 つ目の方法は、コンポーネント要素 (<activity> 、<service> 、<receiver>) の中に <intent-filter> を定義することである。このとき、export タグは自動的に "true" に設定される。他のすべての Android アプリが IPC コンポーネント要素と相互作用するのを防ぐために、これが必要でない限り、android:exported="true" の値と <intent-filter> が彼らの AndroidManifest.xml ファイル内に含まれていないことを確認する。

許可タグ (android:permission を使用すると、他のアプリケーションのコンポーネントへのアクセスも制限されることに注意する。IPC が他のアプリケーションからアクセスされることを意図している場合、<permission> 要素でセキュリティポリシーを適用し、適切な android:protectionLevel を設定することができる。android:permission がサービス宣言で使用される場合、他のアプリケーションは対応する <uses-permission> 要素を自身のマニフェストで宣言し、service を開始、停止、または service に bind する必要がある。

content provider の詳細については、「データストレージのテスト」の章にあるテストケース「 Stored Sensitive Data Is Exposed via IPC Mechanisms 」を参照する。

IPC メカニズムのリストを特定したら、ソースコードをレビューして、そのメカニズムが使用されたときに機密データが漏洩するかどうかを確認する。例えば、content provider はデータベース情報にアクセスするために使用することができ、service はデータを返すかどうかを確認するためにプローブすることができる。Broadcast receivers は、調査されたり傍受されたりすると、機密情報を漏洩する可能性がある。

以下では、2 つのアプリを例に挙げ、脆弱な IPC コンポーネントを特定する例を紹介する。

- Sieve

  参考

  - *Activity* の公開
  - *Service* の公開

- Android Insecure Bank

  参考

- *BroadcastReceiver* の公開

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Static Analysis

ルールブック

- セキュリティを考慮し *IPC* メカニズムを使用する（必須）

6.4.3 動的解析

MobSF を使用して IPC コンポーネントを列挙できる。エクスポートされた IPC コンポーネントを一覧表示するには、APK ファイルをアップロードすると、以下の画面にコンポーネントコレクションが表示される。

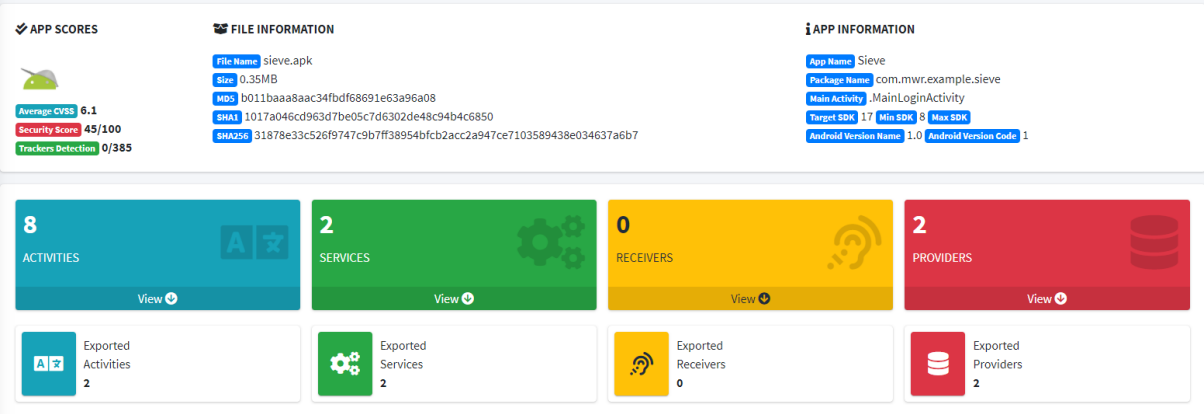


図 6.4.3.1　コンポーネントコレクション

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Dynamic Analysis

6.4.3.1 ContentProvider

Sieve アプリケーションは、脆弱性のある ContentProvider を実装している。Sieve アプリがエクスポートする ContentProvider を一覧表示するには、次のコマンドを実行する。

```
$ 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}
```

特定したら、jadx を使ってアプリをリバースエンジニアリングし、エクスポートされた ContentProvider のソースコードを解析して、潜在的な脆弱性を特定することが可能である。

ContentProvider の対応するクラスを特定するには、次の情報を使用する。

- パッケージ名 : com.mwr.example.sieve
- ContentProvider クラス名 : DBContentProvider

クラス com.mwr.example.sieve.DBContentProvider を分析すると、いくつかの URI が含まれていることがわかる。

```
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");
...
}
```

特定した URI を使用して ContentProvider を呼び出すには、以下のコマンドを使用する。

```
$ 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
```

これで、データベースの全エントリーを取得できるようになった (出力に "Row:" で始まるすべての行が表示される)。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Content Providers

6.4.3.2 Activity

アプリケーションからエクスポートされた activity を一覧表示するには、次のコマンドを使用し、activity 要素にフォーカスを当てる。

```
$ 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
```

(次のページに続く)

(前のページからの続き)

```
  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")
  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
...
```

エクスポートされた activity は、以下のプロパティのいずれかを使用して識別することができる。

- intent-filter のサブ宣言がある。
- android:exported 属性が 0xffffff に設定されている。

また、jadx を使用して、上記の基準を使用して AndroidManifest.xml ファイル内のエクスポートされた activity を識別することもできる。

```
<?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"/>
```

(次のページに続く)

(前のページからの続き)

```
    </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>
```

脆弱性のあるパスワードマネージャー「 Sieve 」の activity を列挙すると、以下の activity がエクスポートされることがわかる。

- .MainLoginActivity
- .PWList
- .FileSelectActivity

activity を起動するには、以下のコマンドを使用する。

```
# 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 }
```

この例では activity .PWList が直接呼び出されているので、パスワードマネージャを保護するログインフォームをバイパスし、パスワードマネージャ内に含まれるデータにアクセスするために使用することができる。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Activities

6.4.3.3 Service

Services は、Drozer のモジュール app.service.info で列挙することができる。

```
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
```

service と通信するには、まず静的解析を使用して必要な入力を特定する必要がある。

この service はエクスポートされているため、app.service.send モジュールを使用して service と通信し、ターゲットアプリケーションに格納されているパスワードを変更することができる。

```
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
```

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Services

6.4.3.4 BroadcastReceiver

アプリケーションによってエクスポートされた broadcast receivers を一覧表示するには、次のコマンドを使用してレシーバー要素に注目する。

```
$ 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")
...
```

次のプロパティのいずれかを使用して、エクスポートされた broadcast receiver を識別できる。

- intent-filter のサブ宣言がある。
- android:exported 属性が 0xffffff に設定されている。

また、jadx を使用して、上記の基準を使用して AndroidManifest.xml ファイル内のエクスポートされた broadcast receiver を識別することができる。

```
<?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>
```

上記の脆弱なバンキングアプリケーション InsecureBankv2 の例では、com.android.insecurebankv2.MyBroadCastReceiver という名前の broadcast receiver だけがエクスポートされていることがわかる。

エクスポートされた broadcast receiver があることがわかったので、さらに深く掘り下げ、jadx を使用してアプリをリバースエンジニアリングすることができる。これにより、ソースコードを解析して、後で悪用できる潜在的な脆弱性を探すことができる。エクスポートされた broadcast receiver のソースコードは、次のとおりである。

```
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();
```

(次のページに続く)

(前のページからの続き)

```
                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();
            }
        } else {
            System.out.println("Phone number is null");
        }
    }
}
```

ソースコードを見ればわかるように、この broadcast receiver は phonenumber と newpass という 2 つのパラメータを想定している。この情報をもとに、カスタム値を使ってイベントを送信することで、この broadcast receiver を悪用することができる。

```
# 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
```

これにより、以下のような SMS が生成される。

```
Updated Password from: SecretPassword@ to: 12345
```

Sniffing Intents Android アプリケーションが、必要なパーミッションの設定や宛先パッケージの指定なしに intents を broadcasts すると、デバイス上で実行されるすべてのアプリケーションによって intent が監視される可能性がある。

broadcast receiver を登録して intents をスニッフィングするには、Drozer モジュール app.broadcast.sniff を使用し、--action パラメータで監視するアクションを指定する。

```
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)`
```

また、以下のコマンドで intents をスニッフィングすることも可能である。ただし、渡された extras の内容は

表示されない。

```
$ 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)
```

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Sniffing Intents

6.4.4 ルールブック

1. セキュリティを考慮し *IPC* メカニズムを使用する（必須）

6.4.4.1 セキュリティを考慮し IPC メカニズムを使用する（必須）

セキュリティを考慮し IPC メカニズムを使用することで、アプリからの機密データの漏洩や公開を未然に防ぐ。

他のアプリへコンポーネントを公開する一般的な方法には、以下の 2 つの方法がある。他のアプリ（外部）へ公開する必要のないコンポーネントは公開しない。

- AndroidManifest.xml のコンポーネントの定義に android:exported="true" を設定する。これにより他のアプリへコンポーネントが公開される。
- AndroidManifest.xml のコンポーネントの定義に <intent-filter> の定義を追加する。これは、API level 31 未満の場合は、定義することで android:exported が自動的に true に設定される。API level 31 以上の場合は、<intent-filter> 定義時には明示的に android:exported を設定する必要がある。設定されていない場合は、インストールに失敗する。

以下は AndroidManifest.xml での <intent-filter> の定義と、android:exported="true" の設定例。

```
<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"
```

(次のページに続く)

(前のページからの続き)

```
        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>
```

また、許可タグ (android:permission) を使用すると、他のアプリケーションのコンポーネントへのアクセスも制限できる。IPC が他のアプリケーションからアクセスされることを意図している場合、<permission> 要素でセキュリティポリシーを適用し、適切な android:protectionLevel を設定することができる。

以下は AndroidManifest.xml でのカスタムパーミッションの定義例。

```
<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>
```

以下は上記で定義されたカスタムパーミッションを他のアプリの AndroidManifest.xml で定義する場合の例。

```
<manifest>
<uses-permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY" />
        <application>
            <activity>
            </activity>
        </application>
</manifest>
```

IPC メカニズムを公開する必要がある場合は、そのメカニズムが使用されたときに機密データを漏洩しないようにソースコードを確認する。

以下のサンプルコードは、Intent へ文字列を設定し、BroadCast を送信する処理である。こういった場合に、文字列に機密データを含まないようにする。

```
Intent().also { intent ->
    intent.setAction("com.example.broadcast.MY_NOTIFICATION")
    intent.putExtra("data", "Notice me senpai!")
    sendBroadcast(intent)
}
```

Intent がブロードキャストされ、アプリケーション内でのみ受信される場合、LocalBroadcastManager を使用して、他のアプリがブロードキャストメッセージを受信するのを防ぐことができる。これにより、機密情報が漏洩するリスクが軽減される。

これに違反する場合、以下の可能性がある。

- IPC メカニズムによる機密データの漏洩や公開に繋がる。

6.5 MSTG-PLATFORM-5

明示的に必要でない限り WebView で JavaScript が無効化されている。

6.5.1 WebView での JavaScript の使用

JavaScript は、反射型、蓄積型、DOM ベースのクロスサイトスクリプティング (XSS) により、ウェブアプリケーションに挿入される可能性がある。モバイルアプリはサンドボックス環境で実行されるため、ネイティブで実装されている場合はこのような脆弱性はない。しかしながら、WebView は、Web ページの閲覧を可能にするために、ネイティブアプリの一部である可能性がある。すべてのアプリは独自の WebView キャッシュを持っており、ネイティブブラウザや他のアプリと共有されることはない。Android では、WebView は WebKit レンダリングエンジンを使用して Web ページを表示するが、ページにはアドレスバーがないなど、最小限の機能に絞り込まれたものとなっている。WebView の実装が甘く、JavaScript の使用が許可されている場合、JavaScript を使用してアプリを攻撃し、そのデータにアクセスすることが可能である。

参考資料

- owasp-mastg Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5) Overview

6.5.1.1 静的解析

WebView クラスの使用方法と実装については、ソースコードを確認する必要がある。WebView を作成し使用するには、WebView クラスのインスタンスを作成する必要がある。

```
WebView webview = new WebView(this);
setContentView(webview);
webview.loadUrl("https://www.owasp.org/");
```

WebView にはさまざまな設定を適用することができる (JavaScript の有効化/無効化がその一例) 。WebView の JavaScript はデフォルトで無効になっているため、明示的に有効にする必要がある。JavaScript の有効化を確認するには、setJavaScriptEnabled というメソッドを探す。

```
webview.getSettings().setJavaScriptEnabled(true);
```

これにより、WebView は JavaScript を解釈することができる。アプリへの攻撃表面を減らすために、必要な場合のみ有効にする必要がある。JavaScript が必要な場合は、以下を確認する必要がある。

- エンドポイントへの通信は一貫して HTTPS (または暗号化が可能な他のプロトコル) に依存し、HTML と JavaScript を伝送中の改ざんから保護する。
- JavaScript と HTML は、アプリのデータディレクトリ内、または信頼できる Web サーバからのみ、ローカルに読み込まれる。
- ユーザが提供する入力に基づき異なるリソースをロードする手段で、どのソースをロードするかをユーザが定義することはできない。

すべての JavaScript ソースコードとローカルに保存されたデータを削除するには、アプリの終了時に clearCache を使用して WebView のキャッシュをクリアする必要がある。

Android 4.4 より古いプラットフォーム (API level 19) を実行するデバイスは、いくつかのセキュリティ上の問題があるバージョンの WebKit を使用している。回避策として、これらのデバイスでアプリを実行する場合は、WebView オブジェクトが信頼できるコンテンツのみを表示することをアプリで確認する必要がある。

参考資料

- owasp-mastg Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5) Static Analysis

ルールブック

- *WebView* での *JavaScript* の使用を制限する（必須）

6.5.1.2 動的解析

動的解析は動作状況に依存する。アプリの WebView に JavaScript を注入するには、いくつかの方法がある。

- エンドポイントにクロスサイトスクリプティングの脆弱性を格納。ユーザが脆弱性のある機能に移動すると、モバイルアプリの WebView にエクスプロイトが送信される。
- 攻撃者は中間者 (MITM) の立場に立ち、JavaScript を注入することでレスポンスを改ざんする。
- マルウェアが、WebView によって読み込まれるローカルファイルを改ざんする。

これらの攻撃ベクトルに対処するために、以下を確認する。

- エンドポイントから提供されるすべての関数に、保存された XSS がないこと。
- アプリのデータディレクトリにあるファイルのみが WebView でレンダリングされること (テストケース「 Testing for Local File Inclusion in WebViews 」参照) 。
- HTTPS 通信は、MITM 攻撃をバイパスするためのベストプラクティスに従って実装される必要がある。これは、以下のことを意味する。
  - すべての通信は TLS によって暗号化されている (テストケース「 Test for Unencrypted Sensitive Data on the Network 」を参照) 。

  - 証明書が適切にチェックされていること (テストケース「Testing Endpoint Identify Verification」参照) 、および/または
  - 証明書が固定されていること (「カスタム証明書ストアと証明書の固定をテストする」を参照) 。

参考資料

- owasp-mastg Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5) Dynamic Analysis

ルールブック

- *WebView* で *JavaScript* 使用時の攻撃ベクトルに対処する（必須）

6.5.2 ルールブック

1. *WebView* での *JavaScript* の使用を制限する（必須）
2. *WebView* で *JavaScript* 使用時の攻撃ベクトルに対処する（必須）

6.5.2.1 WebView での JavaScript の使用を制限する（必須）

すべてのアプリは独自の WebView キャッシュを持っており、ネイティブブラウザや他のアプリと共有されることはない。Android では、WebView は WebKit レンダリングエンジンを使用して Web ページを表示するが、ページにはアドレスバーがないなど、最小限の機能に絞り込まれたものとなっている。WebView の実装が甘く、JavaScript の使用が許可されている場合、JavaScript を使用してアプリを攻撃し、そのデータにアクセスすることが可能である。

そのため、より安全に WebView を使用するためには JavaScript の使用を制限する必要がある。

WebView の JavaScript はデフォルトで無効になっているため、使用するためには明示的に有効にする必要がある。有効化には setJavaScriptEnabled メソッドを使用し、引数に true を設定する必要がある。

```
WebView webview = new WebView(this);
webview.getSettings().setJavaScriptEnabled(true);
setContentView(webview);
webview.loadUrl("https://www.owasp.org/");
```

そのため、JavaScript を有効化する必要がない場合は、setJavaScriptEnabled メソッドで true を設定しないこと。

JavaScript が必要な場合は、以下を確認する必要がある。

- エンドポイントへの通信は一貫して HTTPS (または暗号化が可能な他のプロトコル) に依存し、HTML と JavaScript を伝送中の改ざんから保護する。
- JavaScript と HTML は、アプリのデータディレクトリ内、または信頼できる Web サーバからのみ、ローカルに読み込まれる。
- ユーザが提供する入力に基づき異なるリソースをロードする手段で、どのソースをロードするかをユーザが定義することはできない。

もし、すべての JavaScript ソースコードとローカルに保存されたデータを削除する場合は、アプリの終了時に clearCache を使用して WebView のキャッシュをクリアする。

以下は clearCache による WebView のキャッシュクリアのサンプルコード。

```
webView.clearCache(true);
```

※ Android 4.4 より古いプラットフォーム (API level 19) を実行するデバイスは、いくつかのセキュリティ上の問題があるバージョンの WebKit を使用している。回避策として、これらのデバイスでアプリを実行する場合は、WebView オブジェクトが信頼できるコンテンツのみを表示することをアプリで確認する必要がある。

これに違反する場合、以下の可能性がある。

- JavaScript を使用してアプリを攻撃し、アプリ内のデータにアクセスできる可能性がある。

6.5.2.2 WebView で JavaScript 使用時の攻撃ベクトルに対処する（必須）

アプリの WebView に JavaScript を注入する攻撃ベクトルへ対処するために、以下の対処を行う。

- エンドポイントから提供されるすべての関数に、保存された XSS がないこと。
- アプリのデータディレクトリにあるファイルのみが WebView でレンダリングされること。以下サンプルコードはアプリのデータディレクトリにあるファイルの WebView でのロード方法である。

```
WebView = new WebView(this);
webView.loadUrl("file:///android_asset/filename.html");
```

- HTTPS 通信は、MITM 攻撃をバイパスするためのベストプラクティスに従って実装される必要がある。これは、以下のことを意味する。
  - すべての通信は TLS によって暗号化されている。以下サンプルコードは TLS での通信処理である。

```
    // 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);
```

(次のページに続く)

(前のページからの続き)

```
    // 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);
```

  - 証明書が適切にチェックされていること。証明書の適切なチェックのサンプルコードは「*TrustManager* による検証（必須）」を参照。
  - 証明書が固定されていること。

これに違反する場合、以下の可能性がある。

- ユーザが脆弱性のある機能に移動すると、モバイルアプリの WebView にエクスプロイトが送信される。
- 攻撃者は中間者 (MITM) の立場に立ち、JavaScript を注入することでレスポンスを改ざんする。
- マルウェアが、WebView によって読み込まれるローカルファイルを改ざんする。

6.6 MSTG-PLATFORM-6

WebView は最低限必要のプロトコルハンドラのセットのみを許可するよう構成されている (理想的には、https のみがサポートされている) 。file , tel , app-id などの潜在的に危険なハンドラは無効化されている。

6.6.1 WebView で許可するプロトコルハンドラ

Android の URL には、いくつかのデフォルトスキーマが用意されている。これらは、以下のように WebView 内でトリガーすることができる。

- http(s)://
- file://
- tel://

WebView はエンドポイントからリモートコンテンツを読み込むことができるが、アプリのデータディレクトリまたは外部ストレージからローカルコンテンツを読み込むこともできる。ローカルコンテンツがロードされる場合、ユーザがファイル名やファイルのロードに使用されるパスに影響を与えることができないようにする必要があり、ユーザがロードされたファイルを編集することができないようにする必要がある。

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Overview

ルールブック

- *WebView* では潜在的に危険なプロトコルハンドラは使用しない（推奨）

6.6.2 静的解析

WebView の使用方法については、ソースコードを確認すること。以下の WebView 設定は、リソースアクセスを制御する。

- setAllowContentAccess：コンテンツ URL アクセスにより、WebView はシステムにインストールされたコンテンツプロバイダからコンテンツをロードすることができ、これはデフォルトで有効になっている。
- setAllowFileAccess：コンテンツへの URL アクセスを許可する。WebView 内のファイルアクセスを有効または無効にする。デフォルト値は、Android 10 (API level 29) 以下をターゲットとする場合は true 、Android 11 (API level 30) 以上をターゲットとする場合は false である。これは、ファイルシステムへのアクセスのみを有効または無効にすることに注意する。アセットとリソースへのアクセスは影響を受けず、file:///android_asset と file:///android_res を介してアクセスできる。
- setAllowFileAccessFromFileURLs：ファイルスキーム URL のコンテキストで実行されている JavaScript が、他のファイルスキーム URL からコンテンツにアクセスすることを許可するかどうかを指定する。デフォルト値は、Android 4.0.3 - 4.0.4 (API level 15) 以下では true 、Android 4.1 (API level 16) 以上では false である。
- setAllowUniversalAccessFromFileURLs を指定する。ファイルスキーム URL のコンテキストで実行されている JavaScript が、任意のオリジンからコンテンツにアクセスすることを許可するかどうかを指定します。デフォルト値は、Android 4.0.3 - 4.0.4 (API level 15) 以下では true、Android 4.1 (API level 16) 以上では false である。

上記のメソッドが 1 つ以上有効な場合、そのメソッドがアプリの正常な動作に本当に必要なのかどうかを判断する必要がある。

WebView インスタンスを特定できる場合は、loadURL メソッドでローカルファイルを読み込んでいるかどうかを確認する。

```
WebView = new WebView(this);
webView.loadUrl("file:///android_asset/filename.html");
```

HTML ファイルの読み込み元を確認する必要がある。例えば外部ストレージから読み込んだ場合、そのファイルは誰でも読み書き可能な状態になっている。これはバッドプラクティスと考えられている。代わりに、ファ

イルはアプリの assets ディレクトリに置かれるべきである。

```
webview.loadUrl("file:///" +
Environment.getExternalStorageDirectory().getPath() +
"filename.html");
```

loadURL で指定された URL は、操作可能な動的パラメータがあるかどうかをチェックする必要がある。その操作により、ローカルファイルのインクルージョンが発生する可能性がある。

以下のコードスニペットとベストプラクティスを使用して、該当する場合はプロトコルハンドラを無効にする。

```
//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);
```

- 読み込みを許可するローカルおよびリモートの Web ページとプロトコルを定義するリストを作成する。
- ローカルの HTML/JavaScript ファイルのチェックサムを作成し、アプリの起動中にチェックする。読みにくくするために、JavaScript ファイルを最小化する。

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Static Analysis

ルールブック

- *WebView* では潜在的に危険なプロトコルハンドラは使用しない（推奨）

6.6.3 動的解析

プロトコルハンドラの使用状況を確認するために、アプリの使用中に電話をかける方法やファイルシステムからファイルにアクセスする方法を探す。

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Dynamic Analysis

6.6.4 ルールブック

1. *WebView* では潜在的に危険なプロトコルハンドラは使用しない（推奨）

6.6.4.1 WebView では潜在的に危険なプロトコルハンドラは使用しない（推奨）

WebView ではアプリのデータディレクトリまたは外部ストレージからローカルコンテンツを読み込むことができる。そのため、攻撃者が WebView にスクリプトを挿入できる場合、ローカルリソースにアクセスできてしまい、結果悪意のあるファイルを WebView で読み込む可能性がある。これに対応するために、WebView でのプロトコルハンドラの使用を制限し、ローカルコンテンツの読み込みを防ぐ。したがって、ローカルコンテンツの読み込みが不要な場合は読み込みを無効化するように設定する必要がある。

以下は WebView でのプロトコルハンドラの使用を制限し、ローカルコンテンツの読み込みを防ぐ設定例。

- setAllowContentAccess：コンテンツ URL アクセスにより、WebView はシステムにインストールされたコンテンツプロバイダからコンテンツをロードすることができ、これはデフォルトで有効になっている。
- setAllowFileAccess：コンテンツへの URL アクセスを許可する。WebView 内のファイルアクセスを有効または無効にする。デフォルト値は、Android 10 (API level 29) 以下をターゲットとする場合は true 、Android 11 (API level 30) 以上をターゲットとする場合は false である。これは、ファイルシステムへのアクセスのみを有効または無効にすることに注意する。アセットとリソースへのアクセスは影響を受けず、file:///android_asset と file:///android_res を介してアクセスできる。
- setAllowFileAccessFromFileURLs：ファイルスキーム URL のコンテキストで実行されている JavaScript が、他のファイルスキーム URL からコンテンツにアクセスすることを許可するかどうかを指定する。デフォルト値は、Android 4.0.3 - 4.0.4 (API level 15) 以下では true 、Android 4.1 (API level 16) 以上では false である。
- setAllowUniversalAccessFromFileURLs を指定する。ファイルスキーム URL のコンテキストで実行されている JavaScript が、任意のオリジンからコンテンツにアクセスすることを許可するかどうかを指定します。デフォルト値は、Android 4.0.3 - 4.0.4 (API level 15) 以下では true、Android 4.1 (API level 16) 以上では false である。

以下のサンプルコードは、WebView でのローカルコンテンツの読み込みを防ぐ設定の処理例。

```
//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);
```

また、以下のベストプラクティスに従うこと。

- 読み込みを許可するローカルおよびリモートの Web ページとプロトコルを定義するリストを作成する。
- ローカルの HTML/JavaScript ファイルのチェックサムを作成し、アプリの起動中にチェックする。読みにくくするために、JavaScript ファイルを最小化する。

もし、ローカルストレージからの読み込みが必要な場合は、読み込むファイルコンテンツはアプリの assets ディレクトリに置くこと。外部ストレージのファイルコンテンツは誰でも読み書き可能な状態になっているため。

以下のサンプルコードは、避けるべき外部ストレージからファイルコンテンツを読み込む例であり、このように loadURL で指定された URL が操作可能な動的パラメータが存在しないことをチェックする必要がある。存在する場合、ユーザにより変更可能なローカルファイルのインクルージョンが発生する可能性がある。

```
webview.loadUrl("file:///" +
Environment.getExternalStorageDirectory().getPath() +
"filename.html");
```

これに違反する場合、以下の可能性がある。

- 悪意のあるファイルを WebView で読み込ませ、WebView にスクリプトを挿入する可能性がある。

6.7 MSTG-PLATFORM-7

アプリのネイティブメソッドが WebView に公開されている場合、WebView はアプリパッケージ内に含まれる JavaScript のみをレンダリングしている。

6.7.1 WebView での JavaScript とネイティブとの Bridge

Android では、WebView で実行される JavaScript が、addJavascriptInterface メソッドを使って Android アプリのネイティブ関数 (@JavascriptInterface でアノテーションされている) を呼び出して使用する方法が用意されている。これは、WebView JavaScript Bridge またはネイティブ Bridge と呼ばれている。

addJavascriptInterface を使用すると、その WebView 内に読み込まれるすべてのページに対して、登録された JavaScript Interface オブジェクトへのアクセスを明示的に許可することになることに注意する。つまり、ユーザがアプリやドメインの外に移動した場合、他のすべての外部ページもこれらの JavaScript Interface オブジェクトにアクセスできることになり、これらのインターフェースを通じて機密データが公開されている場合、潜在的なセキュリティリスクが発生する可能性がある。

警告: Android 4.2 (API level 17) 未満の Android バージョンをターゲットとするアプリには、addJavascriptInterface の実装に欠陥があり、リフレクションを悪用した攻撃により、悪意のある JavaScript が WebView に注入されるとリモートコードの実行につながる可能性があるため、十分注意する。これは、Java Object の全てのメソッドがデフォルトでアクセス可能であること (注釈付きのメソッドのみではなく) に起因してる。

参考資料

- owasp-mastg Determining Whether Java Objects Are Exposed Through WebViews (MSTG-PLATFORM-7) Overview

ルールブック

- *WebView JavaScript Bridge* の利用に注意する（必須）

6.7.1.1 静的解析

メソッド addJavascriptInterface が使用されているかどうか、どのように使用されているか、攻撃者が悪意のある JavaScript を注入することができるかどうかを判断する必要がある。

次の例では、addJavascriptInterface が WebView 内の Java Object と JavaScript の橋渡しに使用されている様子を示している。

```
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);
```

Android 4.2 (API level 17) 以降では、@JavascriptInterface というアノテーションにより、JavaScript から Java メソッドへのアクセスを明示的に許可することができる。

```
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();
        }
}
```

このように、JavaScript から returnString メソッドを呼び出すと、文字列「Secret String」が変数 result に格納される。

```
var result = window.Android.returnString();
```

格納された XSS や MITM 攻撃を介して JavaScript コードにアクセスすると、攻撃者は公開された Java メソッドを直接呼び出すことができる。

addJavascriptInterface が必要な場合、以下の点に注意する。

- APK とともに提供される JavaScript のみが、ブリッジの使用を許可されるべきである。例えば、ブリッジされた各 Java メソッドで URL を検証する (WebView.getUrl を使用する)。
- リモートエンドポイントから JavaScript を読み込まないこと。例えば、ページナビゲーションをアプリのドメイン内にとどめ、その他のドメインはデフォルトブラウザ (Chrome、Firefox など) で開くようにする。
- レガシーな理由 (古いデバイスをサポートする必要があるなど) で必要な場合は、少なくともアプリのマニフェストファイルで最小 API level を 17 に設定する（<uses-sdk android:minSdkVersion="17" />)。

参考資料

- owasp-mastg Determining Whether Java Objects Are Exposed Through WebViews (MSTG-PLATFORM-7) Static Analysis

ルールブック

- *WebView JavaScript Bridge* の利用に注意する（必須）

6.7.1.2 動的解析

アプリを動的に解析することで、どの HTML ファイルや JavaScript ファイルが読み込まれ、どのような脆弱性が存在するのかを知ることができる。脆弱性を悪用する手順は、JavaScript のペイロードを生成し、アプリが要求しているファイルに注入することから始る。この注入は、MITM 攻撃や、外部ストレージに保存されている場合は、ファイルの直接修正によって実現できる。すべてのプロセスは、Drozer と weasel (MWR の高度な搾取ペイロード) を介して達成され、フルエージェントのインストール、実行中のプロセスへの制限付きエージェントの注入、リモートアクセスツール (RAT) としてのリバースシェルの接続が可能になる。

この攻撃の詳細な説明は、MWR によるブログ記事に記載されている

参考資料

- owasp-mastg Determining Whether Java Objects Are Exposed Through WebViews (MSTG-PLATFORM-7) Dynamic Analysis

6.7.2 ルールブック

1. *WebView JavaScript Bridge* の利用に注意する（必須）

6.7.2.1 WebView JavaScript Bridge の利用に注意する（必須）

WebView で実行される JavaScript が、addJavascriptInterface メソッドを使って Android アプリのネイティブ関数 (@JavascriptInterface でアノテーションされている) を呼び出して使用する方法が用意されている。

addJavascriptInterface を使用すると、その WebView 内に読み込まれるすべてのページに対して、登録された JavaScript Interface オブジェクトへのアクセスを明示的に許可することになることに注意する必要がある。

※ Android 4.2 (API level 17) 未満の Android バージョンをターゲットとするアプリには、addJavascriptInterface の実装に欠陥があり、リフレクションを悪用した攻撃により、悪意のある JavaScript が WebView に注入されるとリモートコードの実行につながる可能性があるため、十分注意する。

以下のサンプルコードは、Android 4.2 (API level 17) 未満での addJavascriptInterface メソッドによる WebView JavaScript Bridge の例。

```
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);
```

Android 4.2 (API level 17) 以降では、@JavascriptInterface というアノテーションにより、JavaScript から Java メソッドへのアクセスを明示的に許可することができる。以下にサンプルコードを示す。

```
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();
```

(次のページに続く)

(前のページからの続き)

```
        }
}
```

上記のように、JavaScript から returnString メソッドを呼び出すと、文字列「 Secret String 」が変数 result に格納される。

```
var result = window.Android.returnString();
```

もし、addJavascriptInterface が必要な場合、以下の点に注意し使用する。

- APK とともに提供される JavaScript のみが、ブリッジの使用を許可されるべきである。例えば、ブリッジされた各 Java メソッドで URL を検証する (WebView.getUrl を使用する)。
- リモートエンドポイントから JavaScript を読み込まないこと。例えば、ページナビゲーションをアプリのドメイン内にとどめ、その他のドメインはデフォルトブラウザ (Chrome、Firefox など) で開くようにする。
- レガシーな理由 (古いデバイスをサポートする必要があるなど) で必要な場合は、少なくともアプリのマニフェストファイルで最小 API level を 17 に設定する（<uses-sdk android:minSdkVersion="17" />)。

これに違反する場合、以下の可能性がある。

- 登録された JavaScript Interface オブジェクトを想定していない JavaScript によりアクセスされる可能性がある。

6.8 MSTG-PLATFORM-8

オブジェクトのデシリアライゼーションは、もしあれば、安全なシリアライゼーション API を使用して実装されている。

6.8.1 Android でのオブジェクトのシリアライズ

Android でオブジェクトを永続化するには、いくつかの方法がある。

参考資料

- Testing Object Persistence (MSTG-PLATFORM-8) Overview

6.8.1.1 オブジェクトのシリアライズ

オブジェクトとそのデータは、一連のバイトとして表すことができる。これは、オブジェクトのシリアル化を介して Java で行われる。シリアル化は本質的に安全ではない。これは、データを .ser ファイルにローカルに格納するための単なるバイナリ形式（または表現）である。キーが安全に保管されている限り、HMAC でシリアル化されたデータの暗号化と署名が可能である。オブジェクトのデシリアライズには、オブジェクトのシリアライズに使用されたクラスと同じバージョンのクラスが必要である。クラスが変更された後、ObjectInputStream は古い .ser ファイルからオブジェクトを作成できまない。次の例は、Serializable インターフェースを実装して Serializable クラスを作成する方法を示している。

```
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
  //..

}
```

これで、別のクラスで ObjectInputStream/ObjectOutputStream を使用してオブジェクトを読み書きできるようになる。

参考資料

- Testing Object Persistence (MSTG-PLATFORM-8) Object Serialization

6.8.1.2 JSON 形式でのシリアライズ

オブジェクトの内容を JSON に直列化するには、いくつかの方法がある。Android には、JSONObject クラスと JSONArray クラスが付属している。また、GSON 、Jackson 、Moshi など、さまざまなライブラリを使用することができる。各ライブラリの主な違いは、オブジェクトの合成にリフレクションを使うかどうか、アノテーションをサポートしているかどうか、イミュータブルなオブジェクトを生成するかどうか、使用するメモリ量などである。なお、ほとんど全ての JSON 表現は String ベースであり、したがってイミュータブルである。つまり、JSON に格納された秘密は、メモリから削除することが難しくなる。JSON 自体は、(NoSQL) データベースやファイルなど、どこにでも保存することができる。ただ、秘密を含む JSON が適切に保護されていることを確認する必要がある (例：暗号化/ HMAC 化) 。詳しくは「Android のデータストレージ」の章を確認する。GSON を使った JSON の書き方、読み方の簡単な例 (GSON ユーザガイドより) を以下に示す。この例では、BagOfPrimitives のインスタンスの中身を JSON にシリアライズしている。

```
class BagOfPrimitives {
  private int value1 = 1;
```

(次のページに続く)

(前のページからの続き)

```
  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"}
```

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) JSON

ルールブック

- *JSON/XML* 形式でのシリアライズする場合、秘密や変更してはいけない情報の場合には追加の保護を実施し保存する（必須）

6.8.1.3 XML 形式でのシリアライズ

オブジェクトの内容を XML にシリアライズして戻すには、いくつかの方法がある。Android には XmlPullParser というインターフェースがあり、XML のパース処理を簡単に行うことができる。Android には 2 つの実装がある。KXmlParser と ExpatPullParser である。Android Developer Guide には、これらの使い方が詳しく書かれている。次に、Java ランタイムに付属する SAX パーサー など、さまざまな代替手段がある。詳しくは、ibm.com のブログポストを参照する。JSON と同様に、XML もほとんどが String ベースで動作するという問題があり、String 型の秘密はメモリから削除するのが難しくなることを意味する。XML のデータはどこにでも保存できるが (データベース、ファイル)、秘密や変更してはいけない情報の場合には追加の保護が必要である。詳しくは「Android のデータ保存」の章を確認する。先に述べたように、XML の真の危険性は、XML eXternal Entity (XXE) 攻撃にあり、アプリケーション内でアクセス可能な外部データソースを読み取ることを可能にするかもしれないためである。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) XML

ルールブック

- *JSON/XML* 形式でのシリアライズする場合、秘密や変更してはいけない情報の場合には追加の保護を実施し保存する（必須）
- パーサーアプリケーションは外部エンティティの解決を拒否するように構成する（必須）

6.8.1.4 ORM でのシリアライズ

オブジェクトの内容を直接データベースに格納し、データベースの内容でオブジェクトをインスタンス化する機能を提供するライブラリがある。これを ORM (Object-Relational Mapping) と呼ぶ。SQLite データベースを利用するライブラリには以下のものがある。

- OrmLite
- SugarORM
- GreenDAO
- ActiveAndroid

一方、Realm はクラスの内容を保存するために、独自のデータベースを使用する。ORM が提供できる保護の量は、主にデータベースが暗号化されているかどうかに依存する。詳しくは「Android のデータストレージ」の章を確認する。Realm のウェブサイトには、ORM Lite の素敵な例が掲載されている。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) ORM

6.8.1.5 Parcelable によるオブジェクトの取り扱い

Parcelable は、Parcel に書き込んだり、Parcel から復元したりできるインスタンスを持つクラスのためのインターフェースである。Parcel は、Intent の Bundle の一部としてクラスをパックするためによく使われる。以下は、Parcelable を実装した Android 開発者向けドキュメントの例である。

```
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();
```

(次のページに続く)

(前のページからの続き)

```
        }
    }
```

この Parcel と Intent を含む仕組みは時間の経過とともに変化する可能性があり、Parcelable には IBinder ポインタが含まれる可能性があるため、Parcelable 経由でディスクにデータを保存することは推奨されない。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Parcelable

ルールブック

- *Parcelable* 経由でディスクにデータを保存しない（推奨）

6.8.1.6 Protocol Buffers でのシリアライズ

Google が提供する Protocol Buffers は、Binary Data Format によって構造化データをシリアライズするための、プラットフォームや言語に依存しないメカニズムである。CVE-2015-5237 のように、Protocol Buffers にはいくつかの脆弱性が存在する。なお、Protocol Buffers は機密性の保護を提供しない (暗号化は組み込まれていない)。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Protocol Buffers

ルールブック

- *Protocol Buffers* でのシリアライズは控える（推奨）

6.8.2 静的解析

オブジェクトの持続性を利用して機密情報をデバイスに保存する場合、まず情報が暗号化され、署名/ HMAC されていることを確認する。詳細については、「Android におけるデータストレージ」および「Android 暗号化 API 」の章を参照する。次に、復号キーと検証キーは、ユーザが認証された後でのみ取得できるようにする。セキュリティチェックは、ベストプラクティスで定義されているように、正しい位置で実施する必要がある。

常に取ることができる一般的な修正手順がいくつかある。

1. 機密データが暗号化され、シリアライズ/永続化の後に HMAC /署名されていることを確認する。データを使用する前に、署名または HMAC を評価する。詳細は「 Android Cryptographic APIs 」の章を参照する。
2. ステップ 1 で使用したキーが簡単に抜き取れないことを確認する。キーの取得には、ユーザまたはアプリケーションのインスタンスが適切に認証 / 認可されている必要がある。詳細は「Android におけるデータ保存」の章を参照する。
3. デシリアライズされたオブジェクト内のデータが、アクティブに使用される前に慎重に検証されることを確認する (例えば、ビジネス / アプリケーションロジックの悪用がないこと)。

可用性を重視するリスクの高いアプリケーションでは、シリアライズされたクラスが安定している場合にのみ、Serializable を使用することを推奨する。第二に、リフレクションベースの永続化を使用しないことを推奨する。

- 攻撃者は String ベースの引数を通してメソッドのシグネチャを見つけることができる。
- 攻撃者はビジネスロジックを実行するためにリフレクションベースのステップを操作することができるかもしれない。

詳しくは「Android Anti-Reversing Defenses」の章を確認する。

参考資料

- owasp- mastg Testing Object Persistence (MSTG-PLATFORM-8) Static Analysis

ルールブック

- **JSON/XML 形式でのシリアライズする場合、秘密や変更してはいけない情報の場合には追加の保護を実施し保存する（必須）**
- **オブジェクトの持続性を利用して機密情報をデバイスに保存する場合、情報が暗号化され、署名/HMAC する（必須）**
- **可用性を重視するリスクの高いアプリケーションでは、シリアライズされたクラスが安定している場合にのみ、*Serializable* を使用する（推奨）**

6.8.2.1 オブジェクトのシリアライズ

ソースコードから以下のキーワードで検索する。

- import java.io.Serializable
- implements Serializable

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Object Serialization

6.8.2.2 JSON 形式でのシリアライズ

メモリダンプ対策が必要な場合、標準ライブラリではメモリダンプ対策手法の防止を保証できないため、非常に重要な情報が JSON 形式で保存されていないことを確認する。対応するライブラリで、以下のキーワードを確認することができる。

JSONObject : 以下のキーワードでソースコードを検索する。

- import org.json.JSONObject;
- import org.json.JSONArray;

GSON : 以下のキーワードでソースコードを検索する。

- import com.google.gson
- import com.google.gson.annotations
- import com.google.gson.reflect
- import com.google.gson.stream
- new Gson();
- @Expose, @JsonAdapter, @SerializedName,@Since, and @Until などのアノテーションを使用する

Jackson : 以下のキーワードでソースコードを検索する。

- import com.fasterxml.jackson.core
- 古いバージョンでは、import org.codehaus.jackson

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) JSON

ルールブック

- **非常に重要な情報を *JSON* 形式で保存しない（必須）**

6.8.2.3 ORM でのシリアライズ

ORM ライブラリを使用する場合は、データを暗号化されたデータベースに格納し、クラス表現を個別に暗号化して格納する。詳しくは「Android におけるデータ保存」と「Android Cryptographic APIs 」の章を確認する。対応するライブラリで、以下のキーワードを確認することができる。

OrmLite ソースコードから以下のキーワードを検索する。

- import com.j256.*
- import com.j256.dao
- import com.j256.db
- import com.j256.stmt
- import com.j256.table\

ロギングが無効になっていることを確認する。

SugarORM ソースコードから以下のキーワードで検索する。

- import com.github.satyan
- extends SugarRecord<Type>
- AndroidManifest には、DATABASE 、VERSION 、QUERY_LOG 、DOMAIN_PACKAGE_NAME などの値を持つメタデータエントリーがある。

QUERY_LOG が false に設定されていることを確認する。

GreenDAO : 以下のキーワードでソースコードを検索する。

- 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 : 以下のキーワードでソースコードを検索する。

- ActiveAndroid.initialize (<contextReference>);
- import com.activeandroid.Configuration
- import com.activeandroid.query.*

Realm : 以下のキーワードでソースコードを検索する。

- import io.realm.RealmObject;
- import io.realm.annotations.PrimaryKey;

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) ORM

ルールブック

- *ORM* でのシリアライズにおける注意事項（必須）

6.8.2.4 Parcelable によるオブジェクトの取り扱い

Parcelable を含む Bundle を介して機密情報が Intent に格納される場合、適切なセキュリティ対策が取られていることを確認する。明示的な Intent を使用し、アプリケーションレベルの IPC を使用する場合は、適切な追加のセキュリティコントロールを検証する (署名検証、Intent-permissions 、crypto など) 。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Parcelable

ルールブック

- *Parcelable* を含む *Bundle* を介して機密情報が *Intent* に格納される場合、適切なセキュリティ対策を取る（必須）

6.8.3 動的解析

動的解析を行うには、いくつかの方法がある。

1. 実際の永続化の場合 : データストレージの章で説明したテクニックを使用する。
2. リフレクションベースのアプローチの場合 : Xposed を使ってデシリアライズメソッドにフックするか、シリアライズされたオブジェクトに処理できない情報を追加して、それらがどのように処理されるか (例えば、アプリケーションがクラッシュするか、オブジェクトを豊かにすることで余計な情報が抽出されるか) を確認する。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Dynamic Analysis

6.8.4 ルールブック

1. *JSON/XML* 形式でのシリアライズする場合、秘密や変更してはいけない情報の場合には追加の保護を実施し保存する（必須）
2. *Parcelable* 経由でディスクにデータを保存しない（推奨）
3. *Protocol Buffers* でのシリアライズは控える（推奨）
4. オブジェクトの持続性を利用して機密情報をデバイスに保存する場合、情報が暗号化され、署名/*HMAC* する（必須）
5. 可用性を重視するリスクの高いアプリケーションでは、シリアライズされたクラスが安定している場合にのみ、*Serializable* を使用する（推奨）
6. 非常に重要な情報を *JSON* 形式で保存しない（必須）
7. *ORM* でのシリアライズにおける注意事項（必須）
8. *Parcelable* を含む *Bundle* を介して機密情報が *Intent* に格納される場合、適切なセキュリティ対策を取る（必須）

6.8.4.1 JSON/XML 形式でのシリアライズする場合、秘密や変更してはいけない情報の場合には追加の保護を実施し保存する（必須）

JSON/XML のデータはどこにでも保存できる。そのため、外部ストレージに保存されたファイルへアクセスされるリスクから、秘密や変更してはいけない情報を保存する場合は保存場所の検討や暗号化を施す必要がある。

内部ストレージへのデータの保存方法と、暗号化によるデータを保存するサンプルコードについては、以下ルールを参照。

ルールブック

- 機密データはアプリコンテナまたはシステムの資格情報保存機能へ保存する（必須）
- キーマテリアルは、不要になったらすぐにメモリから消去する必要がある（必須）

これに違反する場合、以下の可能性がある。

- 外部ストレージに保存されたファイルへアクセスされ、漏洩する可能性がある。

6.8.4.2 Parcelable 経由でディスクにデータを保存しない（推奨）

Parcel と Intent を含む仕組みは時間の経過とともに変化する可能性があり、Parcelable には IBinder ポインタが含まれる可能性があるため、Parcelable 経由でディスクにデータを保存することは推奨されない。

以下のサンプルコードは Parcelable インターフェースの実装例。

```
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();
```

(次のページに続く)

(前のページからの続き)

```
    }
}
```

これに注意しない場合、以下の可能性がある。

- IBinder ポインタが含まれる可能性がある。

6.8.4.3 Protocol Buffers でのシリアライズは控える（推奨）

Google が提供する Protocol Buffers は、Binary Data Format によって構造化データをシリアライズするための、プラットフォームや言語に依存しないメカニズムである。CVE-2015-5237 のように、Protocol Buffers にはいくつかの脆弱性が存在する。なお、Protocol Buffers は機密性の保護を提供しない (暗号化は組み込まれていない) 。よって、セキュアな実装として利用すべきではない。

※非推奨なルールのため、サンプルコードはなし。

これに注意しない場合、以下の可能性がある。

- 攻撃者によりヒープベースのバッファオーバーフローを引き起こされる可能性がある。
- 機密データを第三者に読み取られる可能性がある。

6.8.4.4 オブジェクトの持続性を利用して機密情報をデバイスに保存する場合、情報が暗号化され、署名/HMAC する（必須）

オブジェクトの持続性を利用して機密情報をデバイスに保存する場合、まず情報が暗号化され、署名/ HMAC されていることを確認する。対応されていない場合は、以下の一般的な修正手順を実施する。

1. 機密データが暗号化され、シリアライズ/永続化の後に HMAC /署名されていることを確認する。データを使用する前に、署名または HMAC を評価する。詳細は「 Android Cryptographic APIs 」の章を参照する。
2. ステップ 1 で使用したキーが簡単に抜き取れないことを確認する。キーの取得には、ユーザまたはアプリケーションのインスタンスが適切に認証 / 認可されている必要がある。詳細は「 Android におけるデータ保存」の章を参照する。
3. デシリアライズされたオブジェクト内のデータが、アクティブに使用される前に慎重に検証されることを確認する (例えば、ビジネス / アプリケーションロジックの悪用がないこと) 。

これに違反する場合、以下の可能性がある。

- オブジェクトのシリアライズにより保存した機密データを第三者により読み取られる可能性がある。

6.8.4.5 可用性を重視するリスクの高いアプリケーションでは、シリアライズされたクラスが安定している場合にのみ、Serializable を使用する（推奨）

可用性を重視するリスクの高いアプリケーションでは、シリアライズされたクラスが安定している場合にのみ、Serializable を使用することを推奨する。第二に、リフレクションベースの永続化を使用しないことを推奨する。

※概念的なルールのため、サンプルコードはなし。

これに注意しない場合、以下の可能性がある。

- 攻撃者は String ベースの引数を通してメソッドのシグネチャを見つけることができる。
- 攻撃者はビジネスロジックを実行するためにリフレクションベースのステップを操作することができる。

6.8.4.6 非常に重要な情報を JSON 形式で保存しない（必須）

メモリダンプ対策が必要な場合、標準ライブラリではメモリダンプ対策手法の防止を保証できないため、非常に重要な情報は JSON 形式で保存しない。対応するライブラリで、以下のキーワードを確認することができる。

JSONObject : 以下のキーワードでソースコードを検索する。

- import org.json.JSONObject;
- import org.json.JSONArray;

GSON : 以下のキーワードでソースコードを検索する。

- import com.google.gson
- import com.google.gson.annotations
- import com.google.gson.reflect
- import com.google.gson.stream
- new Gson();
- @Expose, @JsonAdapter, @SerializedName,@Since, and @Until などのアノテーションを使用する

Jackson : 以下のキーワードでソースコードを検索する。

- import com.fasterxml.jackson.core
- 古いバージョンでは、import org.codehaus.jackson

※非推奨なルールのため、サンプルコードなし。

これに違反する場合、以下の可能性がある。

- メモリダンプにより情報が漏洩する可能性がある。

6.8.4.7 ORM でのシリアライズにおける注意事項（必須）

ORM でのシリアライズを実施する場合、以下に注意する。

- ORM ライブラリを使用する場合は、データを暗号化されたデータベースに格納し、クラス表現を個別に暗号化して格納する。暗号化されたデータベースのサンプルコードは、「*SQLite*」を参照。
- ロギングが無効になっていることを確認する。

これに違反する場合、以下の可能性がある。

- 保存した機密データを第三者により読み取られる可能性がある。
- ログ内容から機密データを第三者により読み取られる可能性がある。

6.8.4.8 Parcelable を含む Bundle を介して機密情報が Intent に格納される場合、適切なセキュリティ対策を取る（必須）

Parcelable を含む Bundle を介して機密情報が Intent に格納される場合、適切なセキュリティ対策が取られていることを確認する。明示的な Intent を使用し、アプリケーションレベルの IPC を使用する場合は、適切な追加のセキュリティコントロールを検証する (署名検証、Intent-permissions 、crypto など) 。

IPC メカニズムと Intent 使用時のセキュリティ対策とサンプルコードについては、「**セキュリティを考慮し *IPC* メカニズムを使用する（必須）**」

これに違反する場合、以下の可能性がある。

- IPC メカニズムを介して機密データを他アプリに読み取られる可能性がある。

7 コード品質とビルド設定要件

7.1 MSTG-CODE-1

アプリは有効な証明書で署名およびプロビジョニングされている。その秘密鍵は適切に保護されている。

7.1.1 アプリ署名時に考慮すべき要素

7.1.1.1 利用する証明書の有効期限

Android では、すべての APK をインストールまたは実行する前に、証明書によるデジタル署名を行うことが義務付けられている。デジタル署名は、アプリケーションの更新時に所有者の身元を確認するために使用される。このプロセスにより、アプリが改ざんされたり、悪意のあるコードが含まれるように変更されたりすることを防ぐことができる。

APK が署名されると、公開鍵証明書が添付される。この証明書は、APK と開発者、および開発者の秘密鍵を一意に関連付ける。アプリをデバッグモードでビルドする場合、Android SDK は、デバッグ専用に作成されたデバッグキーでアプリに署名する。デバッグキーで署名されたアプリは、配布されることを意図しておらず、Google Play ストアを含むほとんどのアプリストアで受け入れられない。

アプリの最終的なリリースビルドは、有効なリリースキーで署名する必要がある。Android Studio では、アプリは手動で署名するか、リリースビルドタイプに割り当てられた署名設定を作成することで署名できる。

Android 9 (API level 28) 以前の Android では、すべてのアプリのアップデートは同じ証明書で署名する必要があるため、25 年以上の有効期間を持つ証明書を推奨している。Google Play で公開されるアプリは、2033 年 10 月 22 日以降に終了する有効期限を持つキーで署名する必要がある。

参考資料

- owasp-mastg Making Sure That the App is Properly Signed (MSTG-CODE-1) Overview

ルールブック

- **アプリの最終的なリリースビルドでは、有効なリリースキーで署名する（必須）**
- **証明書の有効期限（推奨）**

7.1.1.2 アプリの署名スキーム

4 つの APK 署名スキームが利用可能である。

- JAR 署名 (v1 スキーム)
- APK Signature Scheme v2 (v2 スキーム)
- APK Signature Scheme v3 (v3 スキーム)
- APK Signature Scheme v4 (v4 スキーム)

Android 7.0 (API level 24) 以降でサポートされる v2 スキームは、v1 スキームと比較してセキュリティとパフォーマンスが向上している。V3 スキームは、Android 9 (API level 28) 以降でサポートされており、APK アップデートの一部として署名キーを変更する機能をアプリに提供する。この機能は、新旧両方のキーを使用できるようにすることで、互換性とアプリの継続的な可用性を保証する。V4 スキームは、Android 11 (API level 30) 以降でサポートされている。なお、現時点では、apksigner を介してのみ利用可能である。

各署名方式において、リリースビルドは常に以前のすべての方式でも署名されている必要がある。

参考資料

- owasp-mastg Making Sure That the App is Properly Signed (MSTG-CODE-1) Overview

ルールブック

- **アプリの署名スキームによるセキュリティの向上（推奨）**

7.1.2 静的解析

Android 7.0 (API level 24) 以上では v1 と v2 の両方の方式で、Android 9 (API level 28) 以上では 3 つの方式すべてでリリースビルド時に署名されており、APK 内のコード署名証明書が開発者のものであることを確認する。

APK 署名は、apksigner ツールで検証することができる。[SDK-Path]/build-tools/[version] に配置されている。

```
$ 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
```

署名証明書の内容は、jarsigner で確認することができる。なお、デバッグ証明書では Common Name (CN) 属性が "Android Debug" に設定されている。

デバッグ証明書で署名した APK の出力は以下の通りである。

```
$ jarsigner -verify -verbose -certs example.apk

sm     11116 Fri Nov 11 12:07:48 ICT 2016 AndroidManifest.xml

```

(次のページに続く)

(前のページからの続き)

```
      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]
(...)
```

"CertPath not validated" エラーは無視する。このエラーは、Java SDK 7 以上で発生する。jarsigner の代わりに、apksigner により証明書チェーンを検証することができる。

署名の設定は、Android Studio または build.gradle の signingConfig ブロックで管理することができる。v1 および v2 の両方の方式を有効にするには、以下の値を設定する必要がある。

```
v1SigningEnabled true
v2SigningEnabled true
```

アプリをリリースするための設定に関するいくつかのベストプラクティスは、公式の Android 開発者向けドキュメントに記載されている。

最後に、アプリケーションは決して内部のテスト用証明書とともにデプロイされないようにする。

参考資料

- owasp-mastg Making Sure That the App is Properly Signed (MSTG-CODE-1) Static Analysis

ルールブック

- **ターゲットとする *OS* バージョンにマッチした署名を行う（必須）**
- **アプリケーションは決して内部のテスト用証明書とともにデプロイされないようにする（必須）**

7.1.3 動的解析

APL 署名を検証するには、静的解析を使用する必要がある。

参考資料

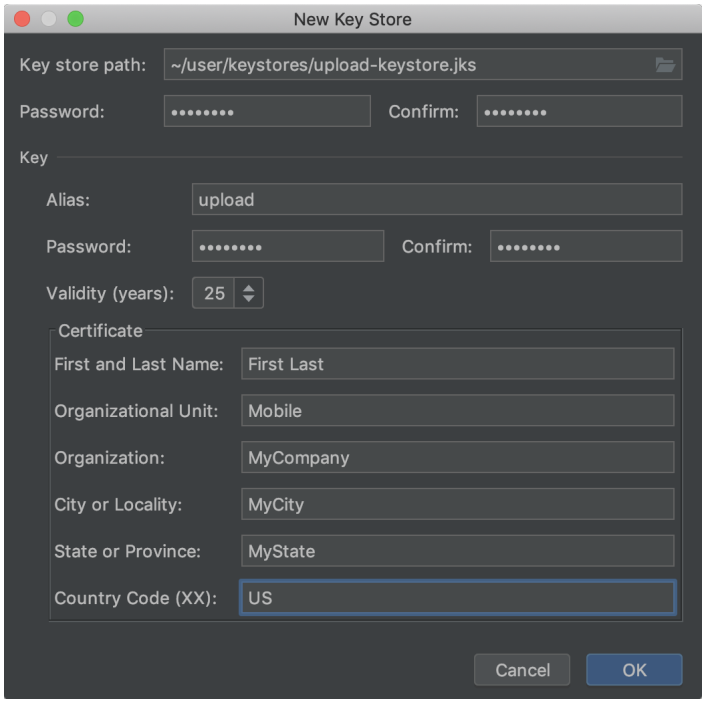
- owasp-mastg Making Sure That the App is Properly Signed (MSTG-CODE-1) Dynamic Analysis

7.1.4 ルールブック

1. **アプリの最終的なリリースビルドでは、有効なリリースキーで署名する（必須）**
2. **証明書の有効期限（推奨）**
3. **アプリの署名スキームによるセキュリティの向上（推奨）**
4. **ターゲットとする *OS* バージョンにマッチした署名を行う（必須）**
5. **アプリケーションは決して内部のテスト用証明書とともにデプロイされないようにする（必須）**

7.1.4.1 アプリの最終的なリリースビルドでは、有効なリリースキーで署名する（必須）

アプリの最終的なリリースビルドは、有効なリリースキーで署名する必要がある。Android Studio では、アプリは手動で署名するか、リリースビルドタイプに割り当てられた署名設定を作成することで署名できる。



参考資料

- アプリに署名して Google Play でリリースする

これに違反する場合、以下の可能性がある。

- Google Play でのアプリの公開ができない。

7.1.4.2 証明書の有効期限（推奨）

Android 9 (API level 28) 以前の Android では、すべてのアプリのアップデートは同じ証明書で署名する必要があるため、25 年以上の有効期間を持つ証明書を推奨している。Google Play で公開されるアプリは、2033 年 10 月 22 日以降に終了する有効期限を持つキーで署名する必要がある。

これに注意しない場合、以下の可能性がある。

- キーの有効期限が切れると、ユーザはアプリの新しいバージョンにシームレスにアップグレードできなくなる。

7.1.4.3 アプリの署名スキームによるセキュリティの向上（推奨）

Android アプリでは以下の 4 つの APK 署名スキームが利用可能である。

- JAR 署名 (v1 スキーム)
- APK Signature Scheme v2 (v2 スキーム)
- APK Signature Scheme v3 (v3 スキーム)
- APK Signature Scheme v4 (v4 スキーム)

Android 7.0 (API level 24) 以降でサポートされる v2 スキームは、v1 スキームと比較してセキュリティとパフォーマンスが向上している。V3 スキームは、Android 9 (API level 28) 以降でサポートされており、APK アップデートの一部として署名キーを変更する機能をアプリに提供する。この機能は、新旧両方のキーを使用できるようにすることで、互換性とアプリの継続的な可用性を保証する。V4 スキームは、Android 11 (API level 30) 以降でサポートされている。なお、現時点では、apksigner を介してのみ利用可能である。

各署名方式において、リリースビルドは常にその前のすべての方式でも署名されている必要がある。等

以下は apksigner による署名方法である。

```
apksigner sign --ks keystore.jks |
  --key key.pk8 --cert cert.x509.pem
  [signer_options] app-name.apk



--v1-signing-enabled <true | false>
指定された APK パッケージに apksigner が署名する際に、従来の JAR ベースの署名スキームを使用するかどうかを指定します。デフォルトでは、このツールは --min-sdk-version と --max-sdk-version の値を使用して、この署名スキームをいつ適用するかを決定します。
--v2-signing-enabled <true | false>
指定された APK パッケージに apksigner が署名する際に、APK 署名スキーム v2 を使用するかどうかを指定します。デフォルトでは、このツールは --min-sdk-version と --max-sdk-version の値を使用して、この署名スキームをいつ適用するかを決定します。
--v3-signing-enabled <true | false>
指定された APK パッケージに apksigner が署名する際に、APK 署名スキーム v3 を使用するかどうかを指定します。デフォルトでは、このツールは --min-sdk-version と --max-sdk-version の値を使用して、この署名スキームをいつ適用するかを決定します。
```

参考資料

- JAR 署名 (v1 スキーム)
- APK Signature Scheme v2 (v2 スキーム)
- APK Signature Scheme v3 (v3 スキーム)
- APK Signature Scheme v4 (v4 スキーム)

これに注意しない場合、以下の可能性がある。

- セキュリティとパフォーマンスが低下する可能性がある。

7.1.4.4 ターゲットとする OS バージョンにマッチした署名を行う（必須）

Android 7.0 (API level 24) 以上では v1 と v2 の両方の方式で、Android 9 (API level 28) 以上では 3 つの方式 (v1, v2, v3) すべてでリリースビルド時に署名する。Android 11 (API level 30) 以上では v4 署名と、これを補完するために v2 または v3 署名が必要である。旧バージョンの Android を実行するデバイスをサポートするには、APK 署名スキーム v2 以降を使用した署名に加えて、引き続き APK 署名スキーム v1 を使用して APK に署名する必要がある。

以下は、apksigner コマンドによる APK への署名方法。

```
apksigner sign --ks [キーストアファイル] -v --ks-key-alias [キーエイリアス] --ks-pass␣
↪pass:[キーストアパスワード] [未署名の APK ファイル]
```

これに違反する場合、以下の可能性がある。

- セキュリティとパフォーマンスが低下する可能性がある。

7.1.4.5 アプリケーションは決して内部のテスト用証明書とともにデプロイされないようにする（必須）

アプリケーションは決して内部のテスト用証明書とともにデプロイされないようにする必要がある。

これに違反する場合、以下の可能性がある。

- ログやデバッグが有効の状態でデプロイされてしまう可能性がある。
- アプリストアで受け入れられない可能性がある。

7.2 MSTG-CODE-2

アプリはリリースモードでビルドされている。リリースビルドに適した設定である（デバッグ不可など）。

7.2.1 アプリのデバッグ有効/無効の切替

Android manifest で定義される Application 要素の android:debuggable 属性は、アプリがデバッグ可能かどうかを決定する。

参考資料

- owasp-mastg Testing Whether the App is Debuggable (MSTG-CODE-2) Overview

7.2.2 静的解析

AndroidManifest.xml を確認し、android:debuggable 属性が設定されているかどうか、またその属性値を確認する。

```
    ...
    <application android:allowBackup="true" android:debuggable="true" android:icon=
↪"@drawable/ic_launcher" android:label="@string/app_name" android:theme="@style/
↪AppTheme">
    ...
```

Android SDK に含まれる apt ツールを以下のコマンドラインで使用すると、android:debuggable="true" ディレクティブが存在するかどうかを迅速に確認することができる。

```
# 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
```

リリースビルドの場合、この属性は常に "false" (デフォルト値) に設定されるべきである。

参考資料

- owasp-mastg Testing Whether the App is Debuggable (MSTG-CODE-2) Static Analysis

ルールブック

- **リリース時は *android:debuggable* 属性を *false* にすべきである（推奨）**

7.2.3 動的解析

adb は、アプリケーションがデバッグ可能かどうかを判断するために使用することができる。

次のコマンドを使用する。

```
# 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
```

デバッグ可能なアプリケーションであれば、アプリケーションコマンドの実行は簡単である。adb shell で、バイナリ名にパッケージ名とアプリケーションコマンドを付加して run-as を実行する。

```
$ 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 は、アプリケーションのデバッグや、アプリのデバッグ有効化の確認にも利用できる。

アプリケーションがデバッグ可能かどうかを判断する別の方法として、実行中のプロセスに jdb をアタッチする方法がある。これが成功すると、デバッグが有効になる。

以下の手順で、jdb を用いたデバッグセッションを開始することができる。

1. adb と jdwp を使用して、デバッグしたいアクティブなアプリケーションの PID を特定する。

```
$ adb jdwp
2355
16346  <== last launched, corresponds to our application
```

2. 特定のローカルポートを使って、アプリケーションプロセス (PID を使用) とホストコンピュータの間に adb による通信チャネルを作成する。

```
# adb forward tcp:[LOCAL_PORT] jdwp:[APPLICATION_PID]
$ adb forward tcp:55555 jdwp:16346
```

3. jdb を使用して、ローカル通信チャネルポートにデバッガをアタッチし、デバッグセッションを開始する。

```
$ jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=55555
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> help
```

デバッグに関するいくつかの注意点は以下の通りである。

- JADX は、ブレークポイント挿入のための興味深い場所を特定するために使用することができる。
- jdb の基本的なコマンドの使い方は、Tutorialspoint に掲載されている。
- jdb がローカル通信チャネルポートにバインドされているときに、「デバッガへの接続が閉じられました」というエラーが発生した場合、すべての adb セッションを終了して、新しいセッションを 1 つだけ開始する。

参考資料

- owasp-mastg Testing Whether the App is Debuggable (MSTG-CODE-2) Dynamic Analysis

7.2.4 ルールブック

1. リリース時は *android:debuggable* 属性を *false* にすべきである（推奨）

7.2.4.1 リリース時は android:debuggable 属性を false にすべきである（推奨）

リリース時は android:debuggable 属性を false にすべてきである。

これに違反する場合、以下の可能性がある。

- 悪意のあるユーザに悪用されてしまう可能性がある。

7.3 MSTG-CODE-3

デバッグシンボルはネイティブバイナリから削除されている。

7.3.1 デバッグシンボルの有無

一般に、コンパイルされたコードには、できるだけ説明を省く必要がある。デバッグ情報、行番号、説明的な関数名やメソッド名などの一部のメタデータは、リバースエンジニアがバイナリやバイトコードを理解しやすくするが、これらはリリースビルドでは必要ないため、アプリの機能に影響を与えることなく安全に省略することが可能である。

ネイティブバイナリを検査するには、nm や objdump などの標準的なツールを使用して、シンボルテーブルを調べる。一般に、リリースビルドにはデバッグ用シンボルを含めるべきではない。ライブラリの難読化が目的であれば、不要なダイナミックシンボルを削除することも推奨される。

参考資料

- owasp-mastg Testing for Debugging Symbols (MSTG-CODE-3) Overview

ルールブック

- リリースビルドではコード情報を出力しないようにする（必須）

7.3.2 静的解析

シンボルは通常ビルドプロセスで取り除かれるので、不要なメタデータが破棄されたことを確認するために、コンパイルされたバイトコードとライブラリが必要である。

まず、Android NDK で nm バイナリを見つけ、それをエクスポートする (またはエイリアスを作成する) 。

```
export NM = $ANDROID_NDK_DIR/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-
↪x86_64/bin/arm-linux-androideabi-nm
```

デバッグシンボルを表示する場合は以下の通りである。

```
$NM -a libfoo.so
/tmp/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin/arm-linux-
↪androideabi-nm: libfoo.so: no symbols
```

ダイナミックシンボルを表示する場合は以下の通りである。

```
$NM -D libfoo.so
```

または、お気に入りのディスアセンブラでファイルを開き、シンボルテーブルを手動でチェックする。

動的シンボルは、visibility コンパイラーフラグによって除去することができる。このフラグを追加すると、gcc は JNIEXPORT として宣言された関数の名前を保持したまま、関数名を破棄するようになる。

build.gradle に以下が追加されていることを確認する。

```
externalNativeBuild {
    cmake {
        cppFlags "-fvisibility=hidden"
    }
}
```

参考資料

- owasp-mastg Testing for Debugging Symbols (MSTG-CODE-3) Static Analysis

7.3.3 動的解析

デバッグ用シンボルの検証には、静的解析を使用する必要がある。

参考資料

- owasp-mastg Testing for Debugging Symbols (MSTG-CODE-3) Dynamic Analysis

7.3.4 ルールブック

1. リリースビルドではコード情報を出力しないようにする（必須）

7.3.4.1 リリースビルドではコード情報を出力しないようにする（必須）

コンパイルされたコードには、できるだけ説明を省く必要がある。デバッグ情報、行番号、説明的な関数名やメソッド名などの一部のメタデータは、リバースエンジニアがバイナリやバイトコードを理解しやすくするが、これらはリリースビルドでは必要ないため、アプリの機能に影響を与えることなく安全に削除する必要がある。

また、リリースビルドにはデバッグ用シンボルを含めるべきではなく、ライブラリの難読化が目的であれば、不要なダイナミックシンボルを削除することも推奨される。

ダイナミックシンボルは、visibility コンパイラーフラグによって除去することができる。このフラグを追加すると、gcc は JNIEXPORT として宣言された関数の名前を保持したまま、関数名を破棄するようになる。

build.gradle に以下が追加されていることを確認する。

```
externalNativeBuild {
    cmake {
        cppFlags "-fvisibility=hidden"
    }
}
```

これに違反する場合、以下の可能性がある。

- コード内のデバッグ情報、行番号、説明的な関数名やメソッド名などの一部のメタデータを漏洩する可能性がある。

7.4 MSTG-CODE-4

デバッグコードおよび開発者支援コード (テストコード、バックドア、隠し設定など) は削除されている。アプリは詳細なエラーやデバッグメッセージをログ出力していない。

7.4.1 StrictMode の利用

StrictMode は、アプリケーションのメインスレッドでの偶発的なディスクやネットワークアクセスなどの違反を検出するための開発者用ツールである。また、パフォーマンスの高いコードの実装など、良いコーディングの実践を確認するためにも使用できる。

以下は、メインスレッドへのディスクアクセスやネットワークアクセスに対するポリシーが有効な StrictMode の例である。

```
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();
 }
```

DEVELOPER_MODE 条件で if 文の中にポリシーを挿入することを推奨する。StrictMode を無効にするには、リリースビルドで DEVELOPER_MODE を無効にする必要がある。

参考資料

- owasp-mastg Testing for Debugging Code and Verbose Error Logging (MSTG-CODE-4) Overview

ルールブック

- **デバッグ時のみ *StrictMode* を使用する（推奨）**

7.4.2 静的解析

StrictMode が有効かどうかを判断するには、StrictMode.setThreadPolicy または StrictMode.setVmPolicy メソッドを探せばよい。ほとんどの場合、これらは onCreate メソッドにある。

スレッドポリシーの検出方法は以下の通りである。

```
detectDiskWrites()
detectDiskReads()
detectNetwork()
```

スレッドポリシー違反の罰則は以下の通りである。

```
penaltyLog() // Logs a message to LogCat
penaltyDeath() // Crashes application, runs at the end of all enabled penalties
penaltyDialog() // Shows a dialog
```

StrictMode を使用するためのベストプラクティスを確認する。

参考資料

- owasp-mastg Testing for Debugging Code and Verbose Error Logging (MSTG-CODE-4) Static Analysis

7.4.3 動的解析

StrictMode を検出する方法はいくつかある。最適な方法は、ポリシーの役割がどのように実装されているかによる。以下のようなものが存在する。

- Logcat
- 警告ダイアログ
- アプリケーションのクラッシュ

参考資料

- owasp-mastg Testing for Debugging Code and Verbose Error Logging (MSTG-CODE-4) Dynamic Analysis

7.4.4 ルールブック

1. デバッグ時のみ *StrictMode* を使用する（推奨）

7.4.4.1 デバッグ時のみ StrictMode を使用する（推奨）

StrictMode はアプリのメインスレッドでの偶発的なディスクやネットワークアクセスなどの違反を検出するための開発者用ツールである。StrictMode 利用する場合は、スレッドポリシーと VM ポリシーを設定する必要がある。そのため、開発時に組み込まれた StrictMode の設定処理がリリース版アプリに組み込まれないために、ポリシーの設定処理の前に分岐を設けるなど、対応する必要がある。

以下に開発用に StrictMode のポリシー設定を組み込んだサンプルコードを示す。

```
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();
 }
```

この場合、以下の可能性がある。

- ディスクアクセス等の情報が漏洩する可能性がある。

7.5 MSTG-CODE-5

モバイルアプリで使用されるライブラリ、フレームワークなどのすべてのサードパーティコンポーネントを把握し、既知の脆弱性を確認している。

7.5.1 サードパーティライブラリ使用時の注意点

Android アプリは、サードパーティーライブラリを利用することが多い。これらのサードパーティライブラリを利用することで、開発者は問題を解決するために書くコードが少なくなり、開発が加速される。ライブラリには 2 つのカテゴリがある。

- テストに使われる Mockito や、特定のライブラリをコンパイルするために使われる JavaAssist のようなライブラリのように、実際のプロダクション・アプリケーションに組み込まれない (あるいは組み込まれるべきではない) ライブラリである。
- Okhttp3 のような、実際のプロダクション・アプリケーション内にパックされているライブラリ。

これらのライブラリは、望ましくない副作用をもたらす可能性がある。

- ライブラリは脆弱性を含んでいる可能性があり、それがアプリケーションを脆弱にする。良い例として、OKHTTP の 2.7.5 より前のバージョンでは、TLS チェーン汚染により SSL ピンニングをバイパスすることが可能であった。
- ライブラリがメンテナンスされなくなったり、ほとんど使われなくなったりして、脆弱性の報告や修正がされなくなる。そのため、脆弱性の報告や修正が行われず、ライブラリを通してアプリケーションに不正なコードや脆弱性のあるコードが含まれる可能性がある。
- 「ライブラリは LGPL2.1 などのライセンスを使用できる。この場合、アプリケーションの作成者は、アプリケーションを使用してそのソースの洞察を要求するユーザに、ソース コードへのアクセスを提供する必要がある。実際、アプリケーションは、ソースコードを変更して再配布できるようにする必要がある。これにより、アプリケーションの知的財産 (IP) が危険にさらされる可能性がある。

この問題は、複数のレベルで発生する可能性があることに注意する。webview で JavaScript を使用する場合、JavaScript のライブラリにもこのような問題がある可能性がある。Cordova 、React-native 、Xamarin アプリのプラグイン/ライブラリも同様である。

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Overview

ルールブック

- **サードパーティライブラリの使用には注意する（推奨）**

7.5.2 静的解析

7.5.2.1 使用するライブラリの脆弱性

サードパーティの依存関係における脆弱性の検出は、OWASP Dependency checker によって行うことができる。これは、dependency-check-gradle のような gradle プラグインを使用することで最もよく行われる。このプラグインを使用するためには、以下の手順を適用する必要がある。以下のスクリプトを build.gradle に追加し、Maven セントラルリポジトリからプラグインをインストールする。

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.owasp:dependency-check-gradle:3.2.0'
    }
}

apply plugin: 'org.owasp.dependencycheck'
```

gradle がプラグインを呼び出したら、実行することでレポートを作成することができる。

```
gradle assemble
gradle dependencyCheckAnalyze --info
```

レポートは、特に設定されない限り、build/reports に置かれる。発見された脆弱性を分析するために、レポートを使用する。ライブラリで見つかった脆弱性から何をすべきかについては、改善策を参照する。

プラグインは、脆弱性フィードをダウンロードする必要があることに注意する。プラグインで問題が発生した場合は、ドキュメントを参照する。

また、Sonatype Nexus IQ 、Sourceclear 、Snyk 、Blackduck など、使用するライブラリの依存関係をよりよくカバーする商用ツールもある。OWASP Dependency Checker または他のツールを使用した場合の実際の結果は、(NDK 関連または SDK 関連) ライブラリの種類によって異なる。

最後に、ハイブリッドアプリケーションの場合、RetireJS で JavaScript の依存性をチェックする必要があることに注意する。同様に、Xamarin の場合は、C# の依存性をチェックする必要がある。

ライブラリが脆弱性を含んでいることが判明した場合、以下の推論が適用される。

- そのライブラリは、アプリケーションと一緒にパッケージされているか。次に、そのライブラリに脆弱性のパッチが適用されたバージョンがあるかどうかをチェックする。もしそうでなければ、その脆弱性が実際にアプリケーションに影響を与えるかどうかをチェックする。もしそうであれば、または将来そうなる可能性があるのであれば、同様の機能を提供し、かつ脆弱性のない代替品を探す。
- そのライブラリは、アプリケーションと一緒にパッケージされていないか。脆弱性が修正されたパッチが適用されたバージョンがあるかどうか確認する。そうでない場合、その脆弱性がビルドプロセスに影響を与えるかどうか確認する。その脆弱性がビルドの妨げになったり、ビルドパイプラインのセキュリティを弱めたりする可能性はないか。そして、その脆弱性が修正されている代替案を探す。

ソースが入手できない場合、アプリを逆コンパイルして、JAR ファイルを確認することができる。Dexguard や ProGuard が適切に適用されている場合、ライブラリに関するバージョン情報は消えていることがよくある。そうでない場合は、与えられたライブラリの Java ファイルのコメントで、非常に多くの情報を見つけることができる。MobSF のようなツールは、アプリケーションに含まれる可能性のあるライブラリの分析に役立つ。もし、コメントや特定のバージョンで使用される特定のメソッドによってライブラリのバージョンを取得できるなら、手動で CVE を調べることができる。

もし、そのアプリケーションが高リスクのアプリケーションであれば、結局は手作業でライブラリを吟味することになる。その場合、ネイティブコードに特有の要件があり、それは「コード品質のテスト」の章に記載さ

れている。その次に、ソフトウェアエンジニアリングのベストプラクティスがすべて適用されているかどうかを吟味するのがよい。

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Detecting vulnerabilities of third party libraries

ルールブック

- ライブラリのアプリの依存関係の解析方法（必須）

7.5.2.2 使用するライブラリのライセンス

著作権法に抵触していないことを確認するためには、License Gradle Plugin のようなライブラリの依存関係を繰り返し確認することができるプラグインを使用するのが一番良い方法である。このプラグインは、次の手順で使用できる。

build.gradle ファイルに以下を追加する。

```
plugins {
    id "com.github.hierynomus.license-report" version"{license_plugin_version}"
}
```

プラグインがピックアップされたら、次のコマンドを使う。

```
gradle assemble
gradle downloadLicenses
```

これで、ライセンスレポートが生成され、サードパーティライブラリで使用されているライセンスを参照するために使用することができる。アプリに著作権表示を含める必要があるかどうか、また、ライセンスの種類によってアプリのコードをオープンソースにする必要があるかどうかを確認するために、ライセンス契約を確認する。

依存関係のチェックと同様に、Sonatype Nexus IQ 、Sourceclear 、Snyk 、Blackduck など、ライセンスをチェックできる商用ツールもある。

注意 : サードパーティのライブラリで使用されているライセンスモデルの意味について疑問がある場合は、法律の専門家に相談する。

ライブラリにアプリケーション知的財産 (IP) をオープンソース化する必要があるライセンスが含まれている場合、同様の機能を提供するために使用できるライブラリの代替品があるかどうかを確認する。

注：ハイブリッドアプリの場合、使用されているビルドツールを確認する：それらのほとんどは、使用されているライセンスを見つけるためのライセンス列挙プラグインを持っている。

ソースが入手できない場合、アプリを逆コンパイルして JAR ファイルを確認することができる。Dexguard や ProGuard が適切に適用されている場合、ライブラリに関するバージョン情報が消えていることがよくある。そうでない場合は、与えられたライブラリの Java ファイルのコメントで、まだ非常に頻繁に見つけることができる。MobSF のようなツールは、アプリケーションに同梱されている可能性のあるライブラリの分析に役

立つ。もし、コメントや特定のバージョンで使用されている特定のメソッドによって、ライブラリのバージョンを取得することができれば、手動で使用されているライセンスを調べることができる。

参考資料

- 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 動的解析

このセクションの動的解析は、ライセンスの著作権が順守されているかどうかを検証する。これは、アプリケーションに、サードパーティライブラリのライセンスが要求する著作権に関する記述がある、about または EULA セクションが必要であることを意味することが多い。

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Dynamic Analysis

ルールブック

- **ライセンスの著作権が順守されているかどうかの検証（必須）**

7.5.4 ルールブック

1. **サードパーティライブラリの使用には注意する（推奨）**
2. **ライブラリのアプリの依存関係の解析方法（必須）**
3. **ライセンスの著作権が順守されているかどうかの検証（必須）**

7.5.4.1 サードパーティライブラリの使用には注意する（推奨）

サードパーティライブラリには以下の欠点があるため、使用する場合は吟味する必要がある。

- ライブラリに含まれる脆弱性。脆弱性が含まれるライブラリを使用すると、ライブラリを通してアプリケーションに不正なコードや脆弱性のあるコードが含まれる可能性があるため注意する。また現時点では脆弱性が発見されていない場合でも今後発見される可能性も存在する。その場合は、脆弱性に対応したバージョンに更新するか、更新バージョンがない場合は使用を控える。
- ライブラリに含まれるライセンス。ライブラリの中には、そのライブラリを使用した場合、使用したアプリのソースコードの展開を求めるライセンスが存在するため注意する。

この問題は、複数のレベルで発生する可能性があることに注意する。webview で JavaScript を使用する場合、JavaScript のライブラリにもこのような問題がある可能性がある。Cordova 、React-native 、Xamarin アプリのプラグイン/ライブラリも同様である。

※サードパーティライブラリの使用注意に関するルールのため、サンプルコードなし。

これに注意しない場合、以下の可能性がある。

- アプリケーションに不正なコードや脆弱性のあるコードが含まれており、悪用される可能性がある。
- サードパーティライブラリに含まれるライセンスにより、アプリのソースコードの展開を求められる可能性がある。

7.5.4.2 ライブラリのアプリの依存関係の解析方法（必須）

サードパーティの依存関係における脆弱性の検出は、OWASP Dependency checker によって行うことができる。これは、dependency-check-gradle のような gradle プラグインを使用することで最もよく行われる。このプラグインを使用するためには、以下の手順を適用する必要がある。以下のスクリプトを build.gradle に追加し、Maven セントラルリポジトリからプラグインをインストールする。

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.owasp:dependency-check-gradle:3.2.0'
    }
}

apply plugin: 'org.owasp.dependencycheck'
```

gradle がプラグインを呼び出したら、実行することでレポートを作成することができる。

```
gradle assemble
gradle dependencyCheckAnalyze --info
```

レポートは、特に設定されない限り、build/reports に置かれる。発見された脆弱性を分析するために、レポートを使用する。ライブラリで見つかった脆弱性から何をすべきかについては、脆弱性の理由を確認する。

プラグインは、脆弱性フィードをダウンロードする必要があることに注意する。プラグインで問題が発生した場合は、ドキュメントを参照する。

また、Sonatype Nexus IQ 、Sourceclear 、Snyk 、Blackduck など、使用するライブラリの依存関係をよりよくカバーする商用ツールもある。OWASP Dependency Checker または他のツールを使用した場合の実際の結果は、(NDK 関連または SDK 関連) ライブラリの種類によって異なる。

最後に、ハイブリッドアプリケーションの場合、RetireJS で JavaScript の依存性をチェックする必要があることに注意する。同様に、Xamarin の場合は、C# の依存性をチェックする必要がある。

ライブラリが脆弱性を含んでいることが判明した場合、以下の理由が適用される。

- そのライブラリは、アプリケーションと一緒にパッケージされているか。次に、そのライブラリに脆弱性のパッチが適用されたバージョンがあるかどうかをチェックする。もしそうでなければ、その脆弱性が実際にアプリケーションに影響を与えるかどうかをチェックする。もしそうであれば、または将来そうなる可能性があるのであれば、同様の機能を提供し、かつ脆弱性のない代替品を探す。
- そのライブラリは、アプリケーションと一緒にパッケージされていないか。脆弱性が修正されたパッチが適用されたバージョンがあるかどうか確認する。そうでない場合、その脆弱性がビルドプロセスに影

響を与えるかどうか確認する。その脆弱性がビルドの妨げになったり、ビルドパイプラインのセキュリティを弱めたりする可能性はないか。そして、その脆弱性が修正されている代替案を探す。

ソースが入手できない場合、アプリを逆コンパイルして、JAR ファイルを確認することができる。Dexguard や ProGuard が適切に適用されている場合、ライブラリに関するバージョン情報は消えていることがよくある。そうでない場合は、与えられたライブラリの Java ファイルのコメントで、非常に多くの情報を見つけることができる。MobSF のようなツールは、アプリケーションに含まれる可能性のあるライブラリの分析に役立つ。もし、コメントや特定のバージョンで使用される特定のメソッドによってライブラリのバージョンを取得できるなら、手動で CVE を調べることができる。

もし、そのアプリケーションが高リスクのアプリケーションであれば、結局は手作業でライブラリを吟味することになる。その場合、ネイティブコードに特有の要件があり、それは「コード品質のテスト」の章に記載されている。その次に、ソフトウェアエンジニアリングのベストプラクティスがすべて適用されているかどうかを吟味するのがよい。

これに違反する場合、以下の可能性がある。

- ライブラリに脆弱性のあるコードが含まれており、悪用される可能性がある。

7.5.4.3 ライセンスの著作権が順守されているかどうかの検証（必須）

主要なライセンスに共通する特徴として、派生ソフトウェアを頒布する際は、利用元 OSS の「著作権」「ライセンス文」「免責事項」などを表示しなければならない点が挙げられる。具体的に「何を」「どこに」「どのように」表示するかは**各ライセンスによって異なる可能性がある**ため、詳細は個別のライセンスを丁寧に確認する必要がある。

例えば MIT License では下記の記述によって「著作権表示」および「ライセンス文」を、「ソフトウェアのすべての複製または重要な部分に記載する」よう定められている。

```
The above copyright notice and this permission notice shall be included in all␣
↪copies or substantial portions of the Software.
```

これに違反する場合、以下の可能性がある。

- アプリの IP をオープンソースにする必要があるライセンスがライブラリに含まれている可能性がある。

7.6 MSTG-CODE-6

アプリは可能性のある例外を catch し処理している。

7.6.1 例外処理の注意点

例外は、アプリケーションが異常な状態やエラー状態に陥ったときに発生する。Java と C++ の両方が例外を発生する可能性がある。例外処理のテストは、アプリケーションが例外を処理し、UI やアプリケーションのロギングメカニズムを通じて機密情報を公開することなく安全な状態に移行することを確認することである。

参考資料

- owasp-mastg Testing Exception Handling (MSTG-CODE-6 and MSTG-CODE-7) Overview

7.6.2 静的解析

アプリケーションを理解するためにソースコードを見直し、異なるタイプのエラー (IPC 通信、リモートサービスの呼び出しなど) をどのように処理しているかを確認する。この段階でチェックすべき事項の例をいくつか挙げる。

- アプリケーションは、例外を処理するためによく設計され、統一されたスキームを使用することを確認する。
- NullPointerException 、IndexOutOfBoundsException 、ActivityNotFoundException 、CancellationException 、SQLException などの標準的な RuntimeException を想定し、Null チェックやバウンドチェックなど適切な処理を行う。RuntimeException の利用可能なサブクラスの概要については、Android 開発者向けドキュメントに記載されている。RuntimeException の子クラスは意図的に throw されるべきであり、その意図は呼び出し側のメソッドによって処理されるべきである。
- すべての non-runtime Throwable に対して、適切な catch ハンドラが存在することを確認し、実際の例外を適切に処理するようにする。
- 例外が発生した場合、アプリケーションは同様の動作をする例外のハンドラを一元的に持つようにする。これは静的なクラスでもかまわない。メソッドに固有の例外については、固有の catch blocks を用意する。
- アプリケーションが例外処理中に機密情報を UI やログステートメントで公開しないようにする。ユーザに問題を説明するために、例外はまだ十分に冗長であることを確認する。
- 高リスクのアプリケーションが扱うすべての機密情報が、finally blocks の実行中に常にワイプされることを確認する。

```
byte[] secret;
try{
    //use secret
} catch (SPECIFICEXCEPTIONCLASS | SPECIFICEXCEPTIONCLASS2 e) {
    // handle any issues
```

(次のページに続く)

(前のページからの続き)

```
} finally {
    //clean the secret.
}
```

catch できない例外のために一般的な例外ハンドラを追加することは、クラッシュが差し迫っているときにアプリケーションの状態をリセットするためのベストプラクティスである。

```
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);
    }

    @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);
        }
    }
}
```

ここで、ハンドラのイニシャライザは、カスタムアプリケーションクラス (例えば、Application を継承したクラス) で呼び出す必要がある。

```
@Override
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);
    MemoryCleanerOnCrash.init();
}
```

参考資料

- owasp-mastg Testing Exception Handling (MSTG-CODE-6 and MSTG-CODE-7) Static Analysis

ルールブック

- 例外/エラー処理の適切な実装と確認事項（必須）
- *catch* できない例外の場合のベストプラクティス（推奨）

7.6.3 動的解析

動的解析にはいくつかの方法がある。

- Xposed を使ってメソッドにフックし、予期しない値で呼び出すか、既存の変数を予期しない値 (例えば NULL 値) で上書きする。
- Android アプリケーションの UI フィールドに予期しない値を入力する。
- アプリケーションの intents 、public providers 、および予期しない値を使用して、アプリケーションと対話する。
- ネットワーク通信やアプリケーションに保存されているファイルを改ざんする。

アプリケーションは決してクラッシュしてはならない。

- エラーから回復するか、継続不可能であることをユーザに知らせることができる状態に移行する。
- 必要であれば、ユーザに適切な行動をとるように指示する (メッセージは機密情報を漏らしてはならない) 。
- アプリケーションで使用されるロギングメカニズムにおいて、いかなる情報も提供しないこと。

参考資料

- owasp-mastg Testing Exception Handling (MSTG-CODE-6 and MSTG-CODE-7) Dynamic Analysis

7.6.4 ルールブック

1. 例外/エラー処理の適切な実装と確認事項（必須）
2. *catch* できない例外の場合のベストプラクティス（推奨）

7.6.4.1 例外/エラー処理の適切な実装と確認事項（必須）

Android で例外/エラー処理を実装する場合は、以下内容を確認する必要がある。

- アプリケーションは、例外を処理するためによく設計され、統一されたスキームを使用することを確認する。
- NullPointerException 、IndexOutOfBoundsException 、ActivityNotFoundException 、CancellationException 、SQLException などの標準的な RuntimeException を想定し、Null チェックやバウンドチェックなど適切な処理を行う。RuntimeException の利用可能なサブクラスの概要については、Android 開発者向けド

キュメントに記載されている。RuntimeException の子クラスは意図的に throw されるべきであり、その意図は呼び出し側のメソッドによって処理されるべきである。

- すべての non-runtime Throwable に対して、適切な catch ハンドラが存在することを確認し、実際の例外を適切に処理するようにする。
- 例外が発生した場合、アプリケーションは同様の動作をする例外のハンドラを一元的に持つようにする。これは静的なクラスでもかまわない。メソッドに固有の例外については、固有の catch blocks を用意する。
- アプリケーションが例外処理中に機密情報を UI やログステートメントで公開しないようにする。ユーザに問題を説明するために、例外はまだ十分に冗長であることを確認する。
- 高リスクのアプリケーションが扱うすべての機密情報が、finally blocks の実行中に常にワイプされることを確認する。

これに違反する場合、以下の可能性がある。

- アプリケーションのクラッシュが発生する。
- 機密情報が漏洩する。

7.6.4.2 catch できない例外の場合のベストプラクティス（推奨）

catch できない例外のために一般的な例外ハンドラを追加することは、クラッシュが差し迫っているときにアプリケーションの状態をリセットするためのベストプラクティスである。

```
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);
    }

    @Override
    public void uncaughtException(Thread thread, Throwable ex) {

            //handle the cleanup here
            //....
```

(次のページに続く)

(前のページからの続き)

```
            //and then show a message to the user if possible given the context

        for (Thread.UncaughtExceptionHandler handler : mHandlers) {
            handler.uncaughtException(thread, ex);
        }
    }
}
```

ここで、ハンドラのイニシャライザは、カスタムアプリケーションクラス (例えば、Application を継承したクラス) で呼び出す必要がある。

```
@Override
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);
    MemoryCleanerOnCrash.init();
}
```

これに注意しない場合、以下の可能性がある。

- アプリケーションのクラッシュが発生する。
- 機密情報が漏洩する。

7.7 MSTG-CODE-7

セキュリティコントロールのエラー処理ロジックはデフォルトでアクセスを拒否している。

※例外処理に関する記載は「7.6.1. 例外処理の注意点」へ纏めて記載するため、本章での記載を省略

7.8 MSTG-CODE-8

アンマネージドコードでは、メモリはセキュアに割り当て、解放、使用されている。

7.8.1 ネイティブコードでのメモリ破損バグ

メモリ破損バグは、ハッカーに人気のある主要なものである。この種のバグは、プログラムが意図しないメモリ位置にアクセスするようなプログラミングエラーに起因する。適切な条件下で、攻撃者はこの挙動を利用して、脆弱なプログラムの実行フローを乗っ取り、任意のコードを実行することができる。この種の脆弱性は、様々な方法で発生する。

メモリ破損を悪用する主な目的は、通常、攻撃者がシェルコードと呼ばれる組み立てられたマシン命令を配置した場所にプログラムフローをリダイレクトすることである。この機能を回避するために、攻撃者は ROP (return-oriented programming) を利用する。このプロセスでは、テキストセグメント内の小さな既存のコードチャンク (ガジェット) を連結し、これらのガジェットが攻撃者にとって有用な機能を実行したり、mprotect を呼び出して攻撃者がシェルコードを格納した場所のメモリ保護設定を変更したりする。

Android アプリは、ほとんどの場合、Java で実装されており、設計上、メモリ破損の問題から本質的に安全である。しかし、JNI ライブラリを利用したネイティブアプリは、この種のバグの影響を受けやすくなっている。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.1 バッファオーバーフロー

特定の操作に対して、割り当てられたメモリの範囲を超えて書き込みを行うプログラミングエラーを指す。攻撃者は、この欠陥を利用して、関数ポインタなど、隣接するメモリにある重要な制御データを上書きすることができる。バッファオーバーフローは、以前はメモリ破壊の最も一般的なタイプの欠陥でしたが、さまざまな要因により、ここ数年はあまり見かけなくなった。特に、安全でない C ライブラリ関数を使用することのリスクについて開発者の間で認識されるようになり、さらに、バッファオーバーフローのバグを捕まえることが比較的簡単になったことが、一般的なベストプラクティスとなっている。しかし、このような不具合がないかどうかをテストする価値はまだある。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.2 Out-of-bounds-access

ポインタの演算にバグがあると、ポインタやインデックスが、意図したメモリ構造 (バッファやリストなど) の境界を超えた位置を参照することがある。アプリが境界外のアドレスに書き込もうとすると、クラッシュや意図しない動作が発生する。攻撃者が対象のオフセットを制御し、書き込まれた内容をある程度操作できれば、コード実行の悪用が可能である可能性が高い。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.3 ダングリングポインタ

これは、あるメモリ位置への参照を持つオブジェクトが削除または解放されたにもかかわらず、オブジェクトポインタがリセットされない場合に発生する。プログラムが後でダングリングポインタを使用して、既に割り当て解除されたオブジェクトの仮想関数を呼び出すと、元の vtable ポインタを上書きして実行を乗っ取ることが可能である。また、ダングリングポインタから参照されるオブジェクト変数や他のメモリ構造の読み書きが可能である。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.4 Use-after-free

これは、解放された (割り当て解除された) メモリを参照するダングリングポインタの特殊なケースを指す。メモリアドレスがクリアされると、その場所を参照していたポインタはすべて無効となり、メモリマネージャはそのアドレスを使用可能なメモリのプールに戻すことになる。このメモリアドレスが再割り当てされると、元のポインタにアクセスすると、新しく割り当てられたメモリに含まれるデータが読み取られたり書き込まれたりする。これは通常、データの破損や未定義の動作につながるが、巧妙な攻撃者は、命令ポインタの制御を活用するために適切なメモリ位置を設定することができる。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.5 整数オーバーフロー

演算結果がプログラマが定義した整数型の最大値を超える場合、整数型の最大値に値が「回り込む」ことになり、必然的に小さな値が格納されることになる。逆に、演算結果が整数型の最小値より小さい場合、結果が予想より大きくなる整数型アンダーフローが発生する。特定の整数オーバーフロー/アンダーフローのバグが悪用可能かどうかは、整数の使われ方によって異なる。例えば、整数型がバッファの長さを表す場合、バッファオーバーフローの脆弱性が発生する可能性がある。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.6 フォーマット文字列の脆弱性

C 言語の printf 関数の format string パラメータに未チェックのユーザ入力が渡されると、攻撃者は '%c' や '%n' などのフォーマットトークンを注入してメモリにアクセスする可能性がある。フォーマット文字列のバグは、その柔軟性から悪用するのに便利である。文字列の書式設定結果を出力してしまうと、ASLR などの保護機能を回避して、任意のタイミングでメモリの読み書きができるようになる。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.2 バッファと整数のオーバーフロー

以下のコードは、バッファオーバーフローの脆弱性が発生する条件の簡単な例を示す。

```
void copyData(char *userId) {
   char  smallBuffer[10]; // size of 10
   strcpy(smallBuffer, userId);
}
```

バッファオーバーフローの可能性を特定するには、安全でない文字列関数 (strcpy 、strcat 、その他「 str 」接頭辞で始まる関数など) の使用や、ユーザ入力を限られたサイズのバッファにコピーするなどの潜在的に脆弱な

プログラミング構造を確認する。安全でない文字列関数のレッドフラグと考えられるのは、以下のようなものである。

- strcat
- strcpy
- strncat
- strlcat
- strncpy
- strlcpy
- sprintf
- snprintf
- gets

また、「 for 」または「 while 」ループとして実装されたコピー操作のインスタンスを探し、長さチェックが正しく実行されていることを確認する。

以下のベストプラクティスが守られていることを確認する。

- 配列のインデックス付けやバッファ長の計算など、セキュリティ上重要な操作に整数変数を使用する場合は、符号なし整数型が使用されていることを確認し、整数の折り返しの可能性を防ぐために前提条件テストを実行する。
- アプリは、strcpy 、その他「 str 」という接頭辞で始まるほとんどの関数、sprint 、vsprintf 、gets などの安全でない文字列関数を使用していないこと。
- アプリに C++ コードが含まれる場合、ANSI C++ 文字列クラスを使用する。
- memcpy の場合、ターゲットバッファが少なくともソースと同じサイズであること、両バッファが重複していないことを確認すること。
- 信頼できないデータがフォーマット文字列に連結されることはない。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Buffer and Integer Overflows

ルールブック

- **バッファオーバーフローを引き起こす安全でない文字列関数を使用しない（必須）**
- **バッファオーバーフローのベストプラクティス（必須）**

7.8.2.1 静的解析

低レベルコードの静的コード解析は、それだけで 1 冊の本が完成するほど複雑なトピックである。RATS のような自動化されたツールと、限られた手動での検査作業を組み合わせれば、通常、低い位置にある果実を特定するのに十分である。しかし、メモリ破損は複雑な原因によって引き起こされることがよくある。例えば、use-after-free バグは、すぐには明らかにならない複雑で直感に反するレースコンディションの結果である可能性がある。一般に、見落とされているコードの欠陥の深い部分に起因するバグは、動的解析またはプログラムを深く理解するために時間を投資するテスターによって発見される。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Static Analysis

7.8.2.2 動的解析

メモリ破壊のバグは、入力ファジングによって発見するのが最適である。自動化されたブラックボックスソフトウェアテストの手法で、不正なデータをアプリケーションに継続的に送信し、脆弱性の可能性がないか調査する。このプロセスでは、アプリケーションの誤動作やクラッシュがないかどうかが監視される。クラッシュが発生した場合、(少なくともセキュリティテスト実施者にとっては) クラッシュを発生させた条件から、攻略可能なセキュリティ上の不具合が明らかになることが期待される。

ファズテストの技術やスクリプト (しばしば「ファザー」と呼ばれる) は、通常、半正確な方法で、構造化された入力の複数のインスタンスを生成する。基本的に、生成された値や引数は、少なくとも部分的にはターゲットアプリケーションに受け入れられるが、無効な要素も含まれており、潜在的に入力処理の欠陥や予期しないプログラムの動作を誘発する可能性がある。優れたファザーは、可能性のあるプログラム実行パスの相当量を公開する (すなわち、高いカバレッジの出力) 。入力は、ゼロから生成する方法 (生成ベース) と、既知の有効な入力データを変異させて生成する方法 (変異ベース) の 2 種類がある。

ファジングの詳細については、OWASP Fuzzing Guide を参照する。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Dynamic Analysis

7.8.3 Java/Kotlin コードでのメモリ破損バグ

Android アプリケーションは、多くの場合、メモリ破損の問題のほとんどが取り除かれた VM 上で実行される。しかし、メモリ破損のバグが存在しないわけではない。例えば、CVE-2018-9522 は、Parcels を使用したシリアライゼーションの問題に関連している。次に、ネイティブコードでは、一般的なメモリ破壊のセクションで説明したような問題がまだ見られる。最後に、BlackHat で示された Stagefright 攻撃のように、サポートするサービスにおけるメモリバグが見られる。

メモリリークもしばしば問題になる。例えば、Context オブジェクトへの参照を Activity 以外のクラスに回した場合や、Activity クラスへの参照をヘルパークラスに回した場合などに起こる。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.3.1 静的解析

探すべき項目はいろいろある。

- ネイティブコード部分があるか。もしあるなら、一般的なメモリ破壊のセクションで指定された問題をチェックする。ネイティブコードは、JNI ラッパー、.CPP / .H / .C ファイル、NDK または他のネイティブフレームワークを使用すると簡単に見つけることができる。
- Java コードや Kotlin コードはあるか。A brief history of Android deserialization vulnerabilities で説明されているような、シリアライズ/デシリアライズの問題を探す。

Java / Kotlin のコードにもメモリリークがある可能性があることに注意する。次のような様々な項目がないか探す。BroadcastReceiver の未登録、Activity や View クラスの静的参照、Context への参照を持つ Singleton クラス、Inner クラスの参照、匿名クラスの参照、AsyncTask の参照、Handler の参照、Threading の誤り、TimerTask の参照などである。詳しくは、以下を確認する。

- Android でメモリリークを回避する 9 つの方法
- Android のメモリリークパターン

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Static Analysis

ルールブック

- **シリアライズ/デシリアライズの問題（推奨）**
- **メモリリークがある可能性がある項目を探す（推奨）**

7.8.3.2 動的解析

実行する様々な手順がある。

- ネイティブコードの場合：Valgrind または Mempatrol を使用して、コードによるメモリ使用量とメモリ呼び出しを分析する。
- Java / Kotlin コードの場合：アプリを再コンパイルし、Squares leak canary で使用する。
- Android Studio の Memory Profiler でリークを確認する。
- シリアライズの脆弱性がないか、Android Java Deserialization Vulnerability Tester で確認する。

参考資料

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Dynamic Analysis

7.8.4 ルールブック

1. バッファオーバーフローを引き起こす安全でない文字列関数を使用しない（必須）
2. バッファオーバーフローのベストプラクティス（必須）
3. シリアライズ/デシリアライズの問題（推奨）
4. メモリリークがある可能性がある項目を探す（推奨）

7.8.4.1 バッファオーバーフローを引き起こす安全でない文字列関数を使用しない（必須）

バッファオーバーフローを引き起こす安全でない文字列関数として以下の関数が存在し、これらの未然に使用を控える必要がある。

- strcat
- strcpy
- strncat
- strlcat
- strncpy
- strlcpy
- sprintf
- snprintf
- gets

※非推奨なルールのため、サンプルコードなし。

これに違反する場合、以下の可能性がある。

- バッファオーバーフローを引き起こす可能性がある。

7.8.4.2 バッファオーバーフローのベストプラクティス（必須）

バッファオーバーフローを引き起こさなために以下のことを確認する。

- 配列のインデックス付けやバッファ長の計算など、セキュリティ上重要な操作に整数変数を使用する場合は、符号なし整数型が使用されていることを確認し、整数の折り返しの可能性を防ぐために前提条件テストを実行する。
- アプリは、strcpy 、その他「 str 」という接頭辞で始まるほとんどの関数、sprint 、vsprintf 、gets などの安全でない文字列関数を使用していないこと。
- アプリに C++ コードが含まれる場合、ANSI C++ 文字列クラスを使用する。

- memcpy の場合、ターゲットバッファが少なくともソースと同じサイズであること、両バッファが重複していないことを確認すること。
- 信頼できないデータがフォーマット文字列に連結されることはない。

これに違反する場合、以下の可能性がある。

- バッファオーバーフローを引き起こす可能性がある。

7.8.4.3 シリアライズ/デシリアライズの問題（推奨）

A brief history of Android deserialization vulnerabilities で説明されているような。問題がある。Android のデシリアライゼーションに関する多数の脆弱性を確認し、Android の IPC メカニズムの欠陥がどのようにして 4 つの異なる悪用可能な脆弱性につながるかを示した。

- 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

これに注意しない場合、以下の可能性がある。

- デシリアライズの脆弱性による攻撃。

7.8.4.4 メモリリークがある可能性がある項目を探す（推奨）

- BroadcastReceiver の未登録
- Activity や View クラスの静的参照
- Context への参照を持つ Singleton クラス
- Inner クラスの参照
- 匿名クラスの参照
- AsyncTask の参照
- Handler の参照
- Threading の誤り
- TimerTask の参照

詳しくは、以下を確認する。

- Android でメモリリークを回避する 9 つの方法
- Android のメモリリークパターン

これに注意しない場合、以下の可能性がある。

- メモリーリークを引き起こす可能性がある。

7.9 MSTG-CODE-9

バイトコードの軽量化、スタック保護、PIE サポート、自動参照カウントなどツールチェーンにより提供されるフリーのセキュリティ機能が有効化されている。

7.9.1 バイナリ保護機能の利用

バイナリ保護機能の存在を検出するために使用されるテストは、アプリケーションの開発に使用される言語に大きく依存する。

一般に、すべてのバイナリをテストする必要があり、これには、メインのアプリ実行ファイルとすべてのライブラリ/依存ファイルが含まれる。しかし、Android では、次に述べるようにメインの実行ファイルは安全であると考えられているため、ネイティブライブラリに焦点を当てる。

Android は、アプリの DEX ファイル (classes.dex など) から Dalvik バイトコードを最適化し、ネイティブ コードを含む新しいファイルを生成する (通常、拡張子は .odex または .oat) 。この Android コンパイル済みバイナリは、Linux と Android がアセンブリコードのパッケージに使用する ELF 形式を使用してラップされる。

アプリの NDK ネイティブライブラリも ELF 形式を使用している。

- PIE (Position Independent Executable)
  - Android 7.0 (API level 24) 以降、メインの実行ファイルに対して PIC コンパイルがデフォルトで有効になっている。
  - Android 5.0 (API level 21) で、PIE を有効にしないネイティブライブラリのサポートが停止され、それ以降はリンカーによって PIE が強制的に実行される。
- メモリ管理
  - ガベージコレクションはメインバイナリに対して実行されるだけで、バイナリ自体には何もチェックすることはない。
  - ガベージコレクションは Android のネイティブライブラリには適用されない。開発者は、適切な**手動メモリ管理**を行う責任がある。「**ネイティブコードでのメモリ破損バグ**」を参照する。
- スタックスマッシングプロテクション
  - Android アプリは、メモリ安全とされる Dalvik バイトコードにコンパイルされる (少なくともバッファオーバーフローを軽減するために) 。Flutter などの他のフレームワークは、その言語 (この場合は Dart) がバッファオーバーフローを軽減する方法のため、スタックカナリアを使用してコンパイルされない。
  - Android ネイティブライブラリでは有効になっているはずであるが、完全に判断するのは難しい。
    - NDK ライブラリは、コンパイラがデフォルトでそれを行うので、有効になっているはずである。

    * その他のカスタム C/C++ ライブラリでは有効になっていないかもしれない。

より詳細に知る：

- Android executable formats
- Android runtime (ART)
- Android NDK
- Android linker changes for NDK developers

参考資料

- 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) とは、主記憶装置のどこかに配置されると、その絶対アドレスに関係なく正しく実行されるコードのことである。PIC は共有ライブラリによく使われ、同じライブラリコードを各プログラムのアドレス空間の中で、他の使用中のメモリ (例えば他の共有ライブラリ) と重ならない位置にロードできるようにする。

参考資料

- owasp-mastg Binary Protection Mechanisms Position Independent Code

7.9.1.2 PIE (Position Independent Executable)

PIE (Position Independent Executable) は、すべて PIC で作られた実行バイナリである。PIE バイナリは、実行ファイルのベースやスタック、ヒープ、ライブラリの位置など、プロセスの重要なデータ領域のアドレス空間の位置をランダムに配置する ASLR (アドレス空間レイアウトランダム化) を有効にするために使用される。

参考資料

- owasp-mastg Binary Protection Mechanisms Position Independent Code

7.9.1.3 メモリ管理

自動参照カウント

ARC (Automatic Reference Counting) は、Objective-C と Swift 専用の Clang コンパイラのメモリ管理機能である。ARC は、クラスのインスタンスが不要になったときに、そのインスタンスが使用しているメモリを自動的に解放する。ARC は、実行時に非同期にオブジェクトを解放するバックグラウンドプロセスがない点で、トレースガベージコレクションとは異なる。

トレースガベージコレクションとは異なり、ARC は参照サイクルを自動的に処理しない。つまり、あるオブジェクトへの「強い」参照がある限り、そのオブジェクトは解放されないということである。強い相互参照は、それに応じてデッドロックやメモリリークを発生させる可能性がある。弱い参照を使ってサイクルを断ち切るかどうかは、開発者次第である。ガベージコレクションとの違いについては、こちらで詳しく解説している。

参考資料

- owasp-mastg Binary Protection Mechanisms Automatic Reference Counting

ガベージコレクション

ガベージコレクション (GC) は、Java/Kotlin/Dart など一部の言語が持つ自動的なメモリ管理機能である。ガベージコレクタは、プログラムによって割り当てられたが、もはや参照されていないメモリ (ガベージとも呼ばれる) を回収しようとする。Android ランタイム (ART) は、GC の改良版を使用している。ARC との違いについては、こちらで詳しく説明している。

参考資料

- owasp-mastg Binary Protection Mechanisms Garbage Collection

手動メモリ管理

ARC や GC が適用されない C/C++ で書かれたネイティブライブラリでは、通常、手動でのメモリ管理が必要である。開発者は、適切なメモリ管理を行う責任がある。手動メモリ管理は、不適切に使用された場合、プログラムにいくつかの主要なクラスのバグ、特にメモリ安全性の侵害やメモリリークを引き起こすことが知られている。

より詳細な情報は、「**ネイティブコードでのメモリ破損バグ**」に記載されている。

参考資料

- owasp-mastg Binary Protection Mechanisms Manual Memory Management

7.9.1.4 スタック破壊保護

スタックカナリアは、リターンポインタの直前のスタックに隠された整数値を保存することで、スタックバッファオーバーフロー攻撃を防ぐのに役立つ。この値は、関数の return 文が実行される前に検証される。バッファオーバーフロー攻撃は、しばしばリターンポインタを上書きし、プログラムフローを乗っ取るために、メモリ領域を上書きする。スタックカナリアが有効な場合、それらも上書きされ、CPU はメモリが改ざんされたことを知ることになる。

スタックバッファオーバーフローは、バッファオーバーフロー (またはバッファオーバーラン) と呼ばれる、より一般的なプログラミングの脆弱性の一種である。スタックには、すべてのアクティブな関数呼び出しの戻りアドレスが含まれているため、スタック上のバッファのオーバーフローは、ヒープ上のバッファのオーバーフローよりも、プログラムの実行を狂わせる可能性が高くなる。

参考資料

- owasp-mastg Binary Protection Mechanisms Stack Smashing Protection

7.9.2 静的解析

アプリのネイティブライブラリをテストして、PIE とスタックスマッシングの保護が有効になっているかどうかを判断する。

radare2 の rabin2 を使ってバイナリ情報を取得することができる。例として、UnCrackable App for Android Level 4 v1.0 APK を使用する。

すべてのネイティブライブラリは、canary と pic の両方が true に設定されている必要がある。

libnative-lib.so の場合が以下である。

```
rabin2 -I lib/x86_64/libnative-lib.so | grep -E "canary|pic"
canary   true
pic      true
```

しかし、libtool-checker.so についてはそうではない。

```
rabin2 -I lib/x86_64/libtool-checker.so | grep -E "canary|pic"
canary   false
pic      true
```

この例では、libtool-checker.so をスタックスマッシングプロテクション付きで再コンパイルする必要がある。

参考資料

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Static Analysis

ルールブック

- *PIE* とスタックスマッシングの保護が有効になっている（必須）

7.9.3 ルールブック

1. *PIE* とスタックスマッシングの保護が有効になっている（必須）

7.9.3.1 PIE とスタックスマッシングの保護が有効になっている（必須）

アプリのネイティブライブラリをテストして、PIE とスタックスマッシングの保護が有効になっているかどうかを判断する。

radare2 の rabin2 を使ってバイナリ情報を取得することができる。例として、UnCrackable App for Android Level 4 v1.0 APK を使用する。

すべてのネイティブライブラリは、canary と pic の両方が true に設定されている必要がある。

libnative-lib.so の場合が以下である。

```
rabin2 -I lib/x86_64/libnative-lib.so | grep -E "canary|pic"
canary   true
pic      true
```

しかし、libtool-checker.so についてはそうではない。

```
rabin2 -I lib/x86_64/libtool-checker.so | grep -E "canary|pic"
canary   false
pic      true
```

この例では、libtool-checker.so をスタックスマッシングプロテクション付きで再コンパイルする必要がある。

これに違反する場合、以下の可能性がある。

- スタックバッファオーバーフローを引き起こす可能性がある。

8 更新履歴

2023-04-01

初版